Received: 17 March 2017 | Revised: 5 October 2017 | Accepted: 13 December 2017
DOI: 10.1002/cpe.4424

RESEARCH ARTICLE

WILEY

KeyValueServe†: Design and performance analysis of a multi-tenant data grid as a cloud service

Anwesha Das[1] | Arun Iyengar[2] | Frank Mueller[1]

[1]North Carolina State University, Raleigh, NC 27606, USA
[2]IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA

Correspondence
Anwesha Das, North Carolina State University, Raleigh, NC 27606, USA.
Email: adas4@ncsu.edu

Funding information
National Science Foundation (NSF), Grant/Award Number: 1217748 and 0958311

Summary

Distributed key-value stores have become indispensable for large-scale cluster applications. Many cloud services have deployed in-memory data grids for their enterprise infrastructures and support multi-tenancy services. However, most services do not offer fine-grained multi-tenant resource sharing. To this front, we present *KeyValueServe*, a low overhead cloud service with features aiding resource management. Results based on Hazelcast, a popular open source data grid, indicate that KeyValueServe can efficiently provide services to tenants without degrading performance. Providing consistent performance to all tenants for fluctuating workloads is still difficult. Performance problems occur at scale with diverse tenant requirements. To address this, the paper provides insights to contention and performance bottlenecks. Through experimental analysis, we uncover scenarios of performance degradation and demonstrate optimized performance via coalescing multiple clients' requests. Our work indicates that a Hazelcast cluster can get congested with multiple concurrent connections when processing client requests, resulting in poor performance. KeyValueServe can reduce the number of parallel connections maintained for client requests, resulting in improved performance.

KEYWORDS

cloud computing, data-grid, in-memory, key-value store, multi-tenancy, NoSQL, performance, quality of service

1 | INTRODUCTION

Innovative key-value stores for data intensive computations have been studied thoroughly in recent times such as Bigtable, Pnuts, Dynamo etc.[1-13] Both academically and commercially, high-performance key-value stores have recently gained substantial attention. Table 1 enumerates the same. While some of these are in-memory, the others are disk-based, comprised of Erlang, C, C++, and Java-based solutions. These applications use at least one if not all of Java Native, ReST, and RPC-style client protocols. *Platform* and *Persistence* in Table 1 indicate the main programming language used to develop the enlisted stores and in what way persistence is enabled in them, respectively. It should be noted that many key-value stores can easily be used by applications written in a wide variety of programming languages. NoSQL stores such as MongoDB and DynamoDB are already hosted on clouds such as Amazon Web Services. Google's Cloud Storage sits on its App Engine, which uses various NoSQL solutions like Python dataStore, MongoDB, Cassandra, and RabbitMQ. Similar services available include Joyent[14] offering Riak[15] and Cloudant[16] hosting CouchDB.[17] They have different pricing models based on their architecture and design. In spite of these available services, fine-grained multi-tenant resource sharing (such as a VM or a distributed data structure) is not well studied. Evaluating the trade-offs of designing an in-memory data grid as a cloud service can be an eye-opener to understand its benefits in the context of multi-tenant performance.

Memcached[11] in particular has been used by researchers[18-24] to solve problems related to key-value stores. Besides commercial usage,* novel key-value stores have been contributed from academia, eg, Silt,[13] Voldemort,[8] HyperDex,[7] and Comet.[9] Sustained performance is a primary concern for tenants accessing such stores under fluctuating workloads especially in the context of cloud computing.

† Key-Value Store as a Service.
* See Related work.[1-3,10,12,14-17,19,25,26]

TABLE 1 Popular key-value stores

Name	Platform	Open Source	Persistence
Bigtable	Java	No	Yes
PNUTS	C++	No	Yes, update logs
Dynamo	Java	No	Pluggable Backing store
MongoDB	C/C++	Yes	Yes
Voldemort	Java	Yes	Pluggable
Hadoop	Java	yes	Local OS file system
HBase	Java	Yes	On-disk
Hypertable	C++	Yes	On-disk
HyperDex	C/C++	Yes	On-disk
COMET	Lua	No	Based on Active Storage Objects
SILT	C++	No	Flash drive based
Cassandra	Java	Yes	Custom on-disk
Memcached	C	Yes	Berkeley DB as backend store
Redis	C	Yes	Snapshots on-disk
CouchDB	Erlang	Yes	On-disk
Hazelcast	Java	Yes	In-memory

1.1 | Challenges and motivation

Key-value stores cater to a combination of read (get) and write (put) requests. Ensuring enhanced throughput for ever increasing numbers of tenants is challenging because:

- Data placement and eviction in such stores is oblivious to external tenant and data characteristics. For example, suppose that tenant A's and B's keys share the same cluster instance. If every time B's data are accessed, there is a high probability that A's data get evicted, then A may experience poor performance due to B.
- Since each get/put task has a relatively short task duration, the average task response time, including scheduling/queuing etc., cannot be very high. Executing such tasks while achieving consistent performance is hard.
- Every operation accesses some previously stored data in a typical key-value store unlike queries such as join or merge. Hence, *co-location of data and computation* is important in such clusters. Moreover, a single cluster instance may be over stressed serving multiple tenants, if that instance happens to store the requested keys of those tenants. In that case, it is difficult to ensure well-balanced and distributed request handling, since other instances may be idle serving no requests.
- Atikoglu et al[22] has indicated an imbalance in data center environments such as workload skews and fluctuating request patterns. A system coping with such fluctuations that tries to deliver the desired throughput encounters resource contention.
- With an increase in the number of instances in a cluster and clients, the available network bandwidth becomes a bottleneck. Hence, performance suffers under high cluster load even with access to sufficient resources. This network inflated delay can be particularly *overwhelming* for low latency operations.
- Based on the workload *size* and *type*, the throughput tends to vary. Ensuring tenant performance is a challenge because of the inherent unpredictability of the workload.

These issues coupled with the maintenance problems constantly faced by existing cloud services have inspired the development of KeyValueServe and our corresponding performance study. Addressing the discussed challenges, our proposed framework handles issues related to the co-location of data and computation, network-inflated delay, tenant starvation, and isolation. Our objectives of improving multi-tenant performance and establishing a better service model for key-value stores conform to the major open problems mentioned by Agrawal et al[27] while discussing the desiderata in the context of cloud computing. We have described them in detail in Section 2.

1.2 | Terminology description

We describe three commonly used terms here to indicate their meanings in the context of this work and avoid misunderstandings.

1.2.1 | Multi-tenancy

Most cloud services[28-30] claim to provide multi-tenancy as a service. In reality, tenants are sharing the overall infrastructure but not individual entities like virtual machines or containers. These coarse-grained solutions provide service simultaneously to several customers without sharing a

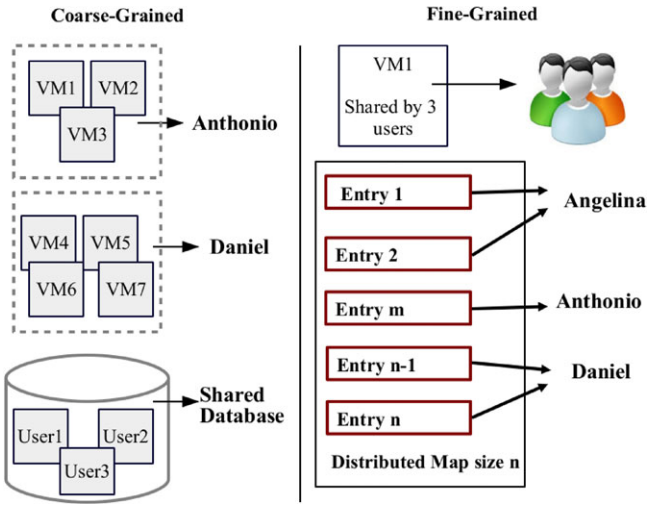

FIGURE 1 Multi-tenant sharing

virtual guest, data structure, or container. Each customer is usually allocated a bunch of VMs for its individual use without any sharing. In our work, we define multi-tenancy as *sharing cloud resources at a finer granularity such as sharing a single instance of the data grid, a single VM part of the key-value store or a single data structure by multiple tenants*. Since in-memory sharing and distributed computing have implications on data structure level sharing, this work discusses a service model pertaining to fine-grained multi-tenancy where unique tenants can seamlessly share a *map* data structure or a cluster *instance* without any inconsistencies. Figure 1 illustrates coarse-grained sharing where users Anthonio and Daniel have dedicated VMs allocated for themselves and the database is shared with three users. Fine-grained sharing is highlighted in the same figure where a single VM and a map data structure are shared by the three users. `Map data structure` in this paper refers to Hazelcast's distributed map API used for the storage and retrieval of keys.

1.2.2 | Throughput degradation

When we refer to performance degradation in this paper, we always refer to *reduced throughput or drop* in terms of operations per second. It should be noted that throughput has been considered from both the server and client's perspective since both the system throughput and per-client throughput indicate the quality of performance. We clarify early on that per-client throughput refers only to the throughput perceived at each client and system throughput refers to the total amount of operations (considering all the clients) serviced from the server's perspective. In the context of multi-tenancy, per-client throughput is important for evaluation. The overall system throughput helps us to understand the system saturation point and gives indications of overload situations.

We reiterate the terms *multi-tenancy* and *performance degradation*; hence, understanding them in the right sense is important.

1.2.3 | Cluster instance

We use the terms cluster instance, server, node, and client frequently. A cluster instance refers to a key-value store instance (Hazelcast in this case) in the cluster, which is a specific JVM (Java Virtual Machine) or an object instance. This need not necessarily refer to a different physical node. These instances can be either servers or clients. Typically, we refer to servers since clients are external instances sending requests to the Hazelcast cluster in this paper.

1.3 | Contributions

With the rise of big data processing and cloud applications, it is understandable that such NoSQL-based solutions enrich cloud computing. These solutions possess their own strengths and weaknesses. In spite of the existence of these solutions, certain facets of Quality of Service (QoS) and performance (eg, fine-grained resource sharing) have not been emphasized enough. Features such as multi-tenancy, novel performance optimization, increased multiplexing with controlled scalability, and flexible pricing models can all together provide a better cloud service. Keeping in mind *service* and *performance*, this paper makes the following contributions.

1. We design and implement *KeyValueServe*, a novel cloud service framework and discuss its architecture.
2. We describe the key features of such a multi-tenant service and demonstrate its negligible overhead.

3. We conduct experiments with the Hazelcast[31] key-value store to observe characteristics and client-performance.
4. We identify causes of performance degradation and develop an approach based on coalescing clients' requests to optimize the performance of tenants. Our optimizations result in more than 10% improvement in the throughput.

KeyValueServe is built on top of Hazelcast to provide a low-overhead service with several helpful features catering to tenant requirements. Our study substantiates the fact that performance degradation is indeed high as the number of clients increases and that an increase in the number of parallel connections with a rising number of clients is one cause for it. We further show how to alleviate performance degradation in Hazelcast through multiplexing multiple client connections maintaining fewer connection instances. Instead of processing every client request individually, we obtained better performance by processing multiple requests, which occur in close temporal proximity to each other in a single batch.

The results from our study can be used to improve performance in similar multi-tenant architectures. The salient features and key takeaways of this work are summarized below.

1. Hazelcast can be used as a cloud service with fine-grained resource sharing such as multi-tenant access to distributed data structures (eg, maps), cluster instances, and virtual machines with negligible overhead (Section 2).
2. JVM-VCPU pinning does not help in reducing contention arising in Hazelcast (Section 3.3).
3. A well-distributed workload across multiple clients improves the perceived response time (Section 3.3).
4. With an increasing number of clients, contention increases and the per-client throughput decreases. However, increasing the number of client side threads does not explicitly degrade performance (Section 3.4).
5. If all the clients have a similar transaction pattern, concurrent data access invariably causes contention no matter where the data is stored (Section 3.5).
6. Read (shared lock used unlike an exclusive lock-based write or update operation) throughput is considerably throttled due to contention. Improving the operational latency is important for a multi-tenant service (Section 3.6).
7. Every new client talking to an instance in the cluster gives rise to an independent connection instance followed by subsequent threads for handling the data read and write operations. Increasing the number of clients results in the creation of more simultaneous parallel connections. Coalescing multiple client requests for processing instead of handling them individually helps in improving the overall per-client performance (Section 3.7).

The rest of this paper is organized as follows. Section 2 presents the KeyValueServe framework for supporting multi-tenancy in cloud services. Section 3 presents the contention analysis and performance optimization. Section 4 surveys previous efforts on cloud-based storage services, efficient performance isolation, and resource sharing, and Section 5 concludes the paper.

2 | KEYVALUESERVE FRAMEWORK

This section proposes KeyValueServe, a cloud based service that can offer a multi-tenant key-value store (Hazelcast) as a service in data centers for efficient computation. However, our ideas are applicable to other key-value stores as well.

2.1 | Hazelcast

Hazelcast[31] is an open source in-memory data grid for distributed computing. Hazelcast's decentralized performance benefits, and its easy deployment make it a good choice for our study. Moreover, it incorporates useful features of co-location of data and computation as well as inherent on-the-fly data redistribution on topology changes, conforming to a strict peer-to-peer model. This makes it suitable for our target stores and aids in addressing the challenges of distributed request handling in a peer-to-peer service. Hazelcast sharding entails equal data and replica distribution on all the instances in the cluster intending to make all nodes fair (in terms of storage and redundancy). The partitions are equally distributed on all the cluster instances (see Section 1.2.3), and a hashing algorithm is used to map data, ie, a key-value item, to a specific partition, ie, an instance.

Figure 2 gives an overview of Hazelcast's threading model. Although several parameters like thread pool size and queue size are configurable, every cluster instance by default has 7 threads serving I/O operations and 5 threads handling events. Additionally, there are dedicated threads to perform partition aware, generic, or priority operations (see the documentation[31] for details). A new client operating in an instance is expensive since several multi-threaded operations are associated with a client. When a client connects to a Hazelcast server instance, the socket acceptor thread establishes communication to process client requests, which are typically various operations such as insert, read, update, etc. Then, subsequent threads are created such as SocketClientDataReader/Writer and read-handler/write-handler to process the requests. These threads are created every time a new client connects to the cluster with varying requests. Thus, with an increasing number of clients per instance, there is an increase in the client threads per instance, and the more the clients talk to members, the higher the internal sharing of data structures across

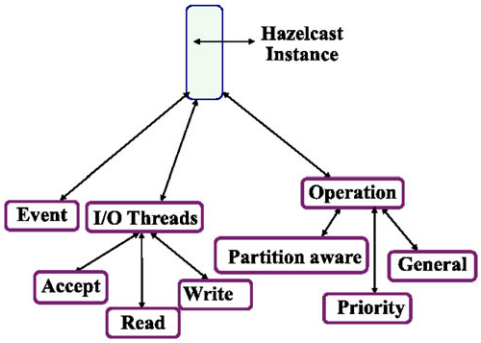

FIGURE 2 Threading model

multiple threads will be. In other words, increasing the number of clients increases internal resource consumption through threads and work queues, which creates an imbalance. Section 3.7 discusses the performance impact of increasing client connections, threads, and their impact on the overall throughput.

2.2 | Framework

This section describes the *KeyValueServe* prototype in more detail. The prototype is built as a decentralized peer-to-peer service model on Hazelcast. The only a priori is that the cluster has to be up and running with at least one instance. Figure 3 shows the primary components of the design.

- *Data nodes*: These are normal data nodes that store data and monitor resource usages sending them periodically to the compute-data nodes.
- *Compute-data nodes*: KeyValueServe chooses to delegate the functionalities of the service layer across multiple Hazelcast nodes without dedicating any single node for the same. This prevents single points of failure and facilitates efficient resource utilization across the nodes, which share data and execution functionalities. In other words, a subset of all the data nodes acts as privileged nodes, which not only store distributed data but also perform some core services. These nodes contain the resource usage statistics and information about all the nodes of the cluster. Such privileged service nodes are called the compute-data nodes. These nodes perform services to enable KeyValueServe features we proposed in Section 2.3. These services are different from the inherent computation available in Hazelcast or any such stores where nodes perform both storage and computation. The idea is to have a peer-to-peer *cloud service model* as well apart from the native peer-to-peer computation and storage model of these stores (with compute and data nodes). This not only fits in the inherent peer-to-peer model of these stores but also aids service resilience without a single point of failure since multiple compute-nodes are enabled to perform a specific service.
- *External client*: Any node that connects to any Hazelcast instance in the cluster not being part of the cluster itself is a client. There may or may not be a dedicated client node (Hazelcast offers different client interfaces where a client performs computations without itself being a part of the cluster).

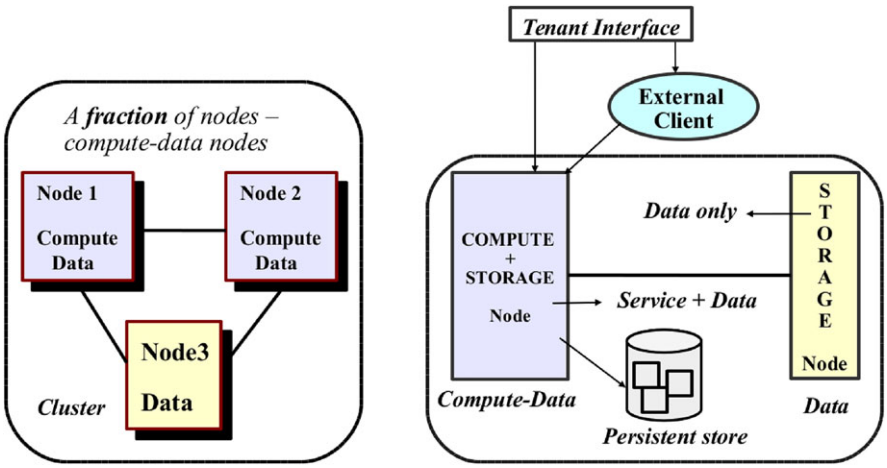

FIGURE 3 Components of KeyValueServe

- *Tenant interface*: External tenants provide input specifications and other requests through the exposed tenant interface to one of the compute-data nodes or the external client, which in turn provides the necessary information to the cluster. Data nodes alone cannot suffice for this interface as services reside on the compute-data nodes.
- *Service resilience*: Distribution of services across multiple compute-data nodes to avoid single points of failure and delegation of equal responsibilities to compute-data nodes instead of over-burdening a single node makes KeyValueServe resilient. The inherent peer-to-peer model as mentioned in Section 2.1 helps in restoring the service to another available data node when a compute-data node goes down. For a stateful service, storing service objects in distributed stores for future restoration on the node failure is a legitimate way to revert back to the same juncture of the service when failures occur. Although a fair distribution is expected, making all the data nodes do some computation may not be a good idea. This is because the cluster size will be large in any data center compared to the distinct services offered, and services starting or stopping on a node when the nodes keep leaving or joining a dynamic cluster are a problem. Hence, only a subset of the cluster should be used as compute-data nodes, as shown in Figure 3. The fraction of compute-data nodes in a cluster is a function of the cluster size and the number of functionalities and features offered by the service. In other words, resilience is enabled by service restoration on another peer, when a specific peer goes down, both configured with the same functionality. As most NoSQL stores have a peer-to-peer model, this service resilience can be applicable to most stores, preventing complete service failures due to the node failures.

The next section discusses the major models of the service distinguishing KeyValueServe from other existing services.

2.3 | Design

The goal of *KeyValueServe* as shown in Figure 4 is to offer additional valuable services in accordance with the normal requirements of a data center (eg, availability, performance, and scalability). Figure 5 illustrates that the service layer sits on top of the key-value store and uses the exposed store APIs to implement the features of the service and to access the store. It aims to enable tenants to efficiently utilize the key-value store with a fine-grained shared storage model, where the degree of multi-tenancy is high. This brings in the following design considerations as part of the service as shown in Figure 6.

- *Multi-tenancy model*: How are multiple tenants going to access the shared storage? Maintaining *tenant-ids* pertaining to specific clients that use the key-value store will enable *tenant authentication*. Access permissions are granted based on this unique tenant identifier. Authentication and controlled access privileges together enhance the multi-tenant sharing of distributed storage enabling opportunities of key-level concurrency. In other words, key-value entries in the distributed data structures such as a map can be shared by multiple tenants. However, ensuring explicit unlocking of keys for exclusive operations is important to avoid tenant starvation at such an increased level of sharing.
- *Transactional model*: Transactional service is required for updates in a concurrent distributed system (such as additions done on maps, queues, etc). Once a tenant is authorized, it can rollback or commit a transaction on its data. This is a feature where users need not worry about the "under-the-hood" implementation specifics for transaction operations.
- *Garbage collection*: In addition to the default garbage collection, the service keeps track of data freshness based on the recency of access and tenant activity. Storage flushing is performed based on infrequent data access during tenant inactivity. If users do not access their data for a considerable period of time (configurable parameter as a threshold), then that data gets stale because no one accessed it for the specified amount of time. It is better to replace that data with recent data accessed by active tenants. Active tenants are those who have not been inactive in terms of reading/writing/updating their data in the recent past. This feature considers both recency and frequency to replace or refill storage. This is

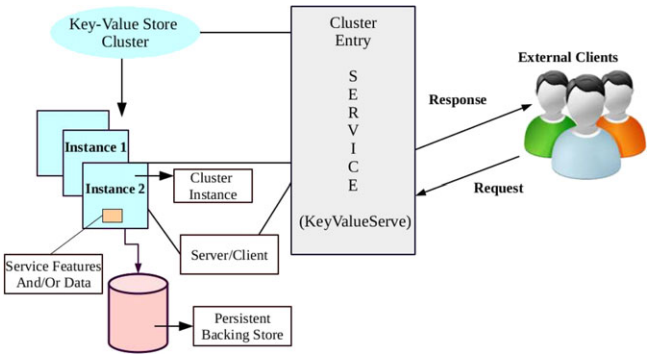

FIGURE 4 KeyValueServe Framework

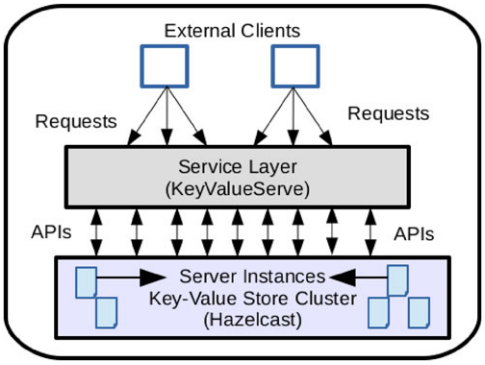

FIGURE 5 Service over key-value store

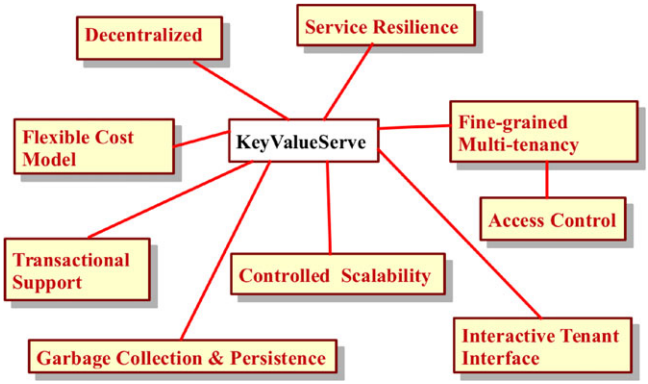

FIGURE 6 Salient service features

different from the built-in JVM garbage collection, which handles automatic memory deallocation for Java objects. During flushing, KeyValueServe either spills to the disk for *persistence* needed for future use or deletes the user's data forever. This makes room for fresh data in similar *in-memory* (non-disk-based) stores where insufficient RAM is often a performance bottleneck. This customized garbage collection scheme also eliminates the need to start additional nodes in the cluster for storing a tenant's data when the overall cluster memory size gets insufficient. KeyValueServe uses the `time-to-live` field of *map* configuration, which enables the eviction of a map entry after the specified time in seconds has elapsed. This addresses the challenge of biased eviction of a tenant's keys without considering how active the tenant is. This feature ensures that an active tenant's keys are retained consciously and less active tenant's keys are evicted, freeing up space. This enables better performance for frequent users.

- *Persistence*: Interfaces such as *MapLoad* and *MapStore* are exposed for data persistence. Persistence avoids the risk of data loss or corruption, which arises when relying solely on in-memory storage, by storing data on the disk. All tenants are paying for their data. Now, the question is *when does it make sense to store data persistently?* When the data structures reach their full capacity they can be stored in an external database. In addition to that there can be three circumstances that require for persistence.

  - When crucial/sensitive data need to be retained, which cannot be lost at any cost, in this case, we need the consent of the tenant.
  - When data are rarely accessed, in-memory storage is not a necessity. That space can be utilized for some other tenant's data. Based on access frequency, the data are spilled to disk for effective memory utilization.
  - When tenants are willing to *pay more* for persistent storage exclusively for their data, which will be a rare situation since the idea behind such stores is efficient in-memory sharing.

  KeyValueServe follows the above persistence model to enable flexible tenant data storage and retention.
- *Cost model*: In the cloud pay-as-you-go model, cost is based on the amount of resources consumed, the duration of usage, and the granularity of access privileges provided to a tenant. The last factor is hard to define since more freedom implies more cost, and the definition of freedom may be different for different tenants. Our cost model charges the tenants based on their *amount of resource consumption* and *duration* of usage. During peak hours, both *amount* and *duration* are considered as opposed to off-peak hours when only duration is considered. This is a weighted average over multiple resources. However, if a tenant expects to use a VM or a map dedicated to itself, through reservation, he needs to pay more since it is free from sharing and consistency issues. Furthermore, VMs are charged higher than maps since the more coarse-grained resource

or entity sharing is (ie, more resource allocation), the higher the cost is. To retain reservations over a period of time, *nominal charges* are made. To summarize, the amount of resources used, the duration of usage, the sharing level (shared versus dedicated), and the granularity (VM versus physical node versus distributed map) govern this cost model. No upfront costs or long-term commitments are required. Illustrative example: say *memory* is charged $0.065 per hour and $0.08 per 500MB and a user uses 1GB of memory for 3 hours. As per this cost model, the *peak* hour cost is $(2 * 0.08) + (3 * 0.065) = \$0.355$ and the *off-peak* hour cost is $(3 * 0.065) = \$0.195$.

- *Controlled scalability*: Increasing the number of cluster instances increases network congestion and the overall resource consumption in the cluster. KeyValueServe aims to multiplex cluster resources as efficiently as possible. This generates the need for a threshold of resource consumption based on which a number of new instances are started or existing instances are terminated. The compute-data nodes collect resource usage statistics from the cluster instances and check for a violation of this threshold. As an example, let the threshold be defined in terms of the CPU consumption to be maintained; suppose this threshold is between 50% and 90%. This implies that if the overall CPU consumption of the cluster goes beyond 90%, a new server instance needs to be created for handling tenant requests. However, if the overall cluster load is below 50%, some instances can be shutdown. This is applicable for multiple types of resources and controls the size of the cluster based on the resources consumed depending on the tenant workload. Overload violation checks based on a specified threshold trigger termination or instantiation of cluster instances. Based on cluster load, the size of the cluster is dynamically varied. Consolidation of services by shutting down instances or starting new instances can help the load balancing in the presence of fluctuating workload conditions. This works well in this data grid since data re-distribution is taken care of by Hazelcast on-the-fly.

To put it into perspective, a multi-tenancy model in conjunction with garbage collection addresses the challenges of tenant starvation and isolation in fine-grained sharing. The challenge of network-induced congestion and performance optimization of multiple tenants is handled in Section 3.7 to enhance the client throughput. Existing services may have considered a subset of the aforementioned features, but a holistic service catering to similar functionalities has not been previously presented to the best of our knowledge.

2.4 | Implementation

A prototype of *KeyValueServe* was built using Hazelcast version 3.3. Python and bash scripts have been used for coding the service layer. Various APIs and interfaces were used to implement the service (eg, utilities such as `forceUnlock` and `TransactionContext` have been used to prevent tenant starvation and enable transactional service support). We deliberately focus on our contribution in designing the key service models in the context of a modern cloud service instead of detailing how the built-in features of the key-value store are leveraged. The latter will vary based on the underlying store used. Our experiments indicated no additional service layer overhead.

2.5 | Evaluation

Results in this section show that the KeyValueServe service layer has a negligible impact on performance. The YCSB[32] benchmarking tool was used to generate load on the system. Experiments were conducted on Fedora and CentOS-based VMs hosted on the IBM RC2 (Research Compute Cloud) cluster, [33] a cloud computing platform used by IBM Research. As mentioned in Section 1.2.2, both the per-client throughput and system throughput have been assessed in the results. Parameters such as the number of clients and the workload type used in the experiments, if varied, have been identified. The environments do not change in any other way. We have observed that even if the experiments are repeated, the throughput numbers do not vary much when the system configuration parameters are kept constant. This indicates a stable well warmed-up system.

The overall *runtime*, *throughput*, and *average latency* are shown in Figures 7 to 12. *Workloada* consists of 50% get and 50% put requests for objects of size 100 000. Ten threads with 100 ops/sec as the target throughput are used for the experimental runs. A target of 10 ops/sec means each YCSB client aims to perform 10 operations per second. YCSB allows us to set target throughput using the `target` parameter. This option determines the number of operations per second based on the supplied value of `target` throughput. We use this `target` parameter to enforce the target throughput in our evaluation; accordingly, ops/sec gets throttled. Each YCSB client can function as a single worker. Specifying 10 threads enables 9 additional workers for each client. This increases the load offered. A combination of the target throughput and number of client threads is used to fine-tune the workload. The load phase of the workload inserts data into the store. In Figures 7 to 12, the *transaction* phase throughput is shown, which indicates the operations executed on that inserted data such as read, write etc. These Figures refer to the client side *read* and *update latencies* (perceived at the YCSB client) using `Workloada`, not to be confused with the additional activity (Section 2.5.1) performed by the store/service such as spawning new instances, map updates, and evictions. The Y axis in these Figures pertaining to Hazelcast activity refers to the following.

- NoWork - This refers to the naive case where a Hazelcast instance is up and running doing nothing as part of the activity. It serves requests of the clients internally, but there are no additional activities going on as part of the cluster or service (no explicit action).

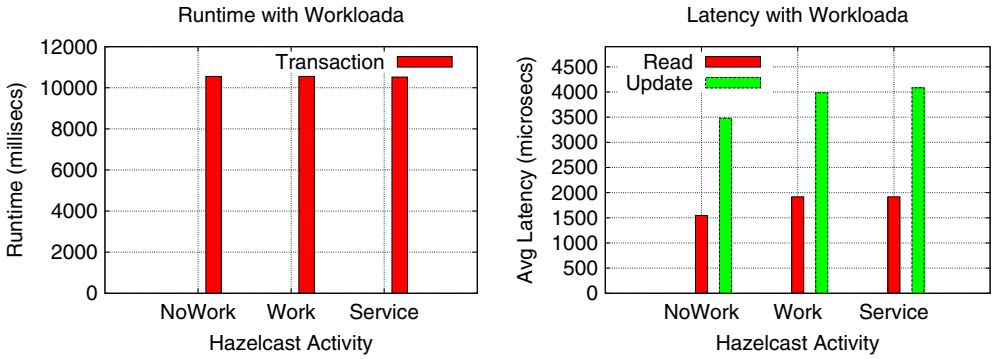

FIGURE 7 Scalability evaluation (runtime and latency)

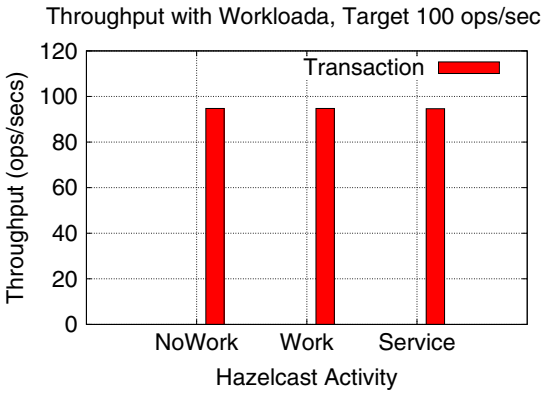

FIGURE 8 Scalability evaluation (throughput)

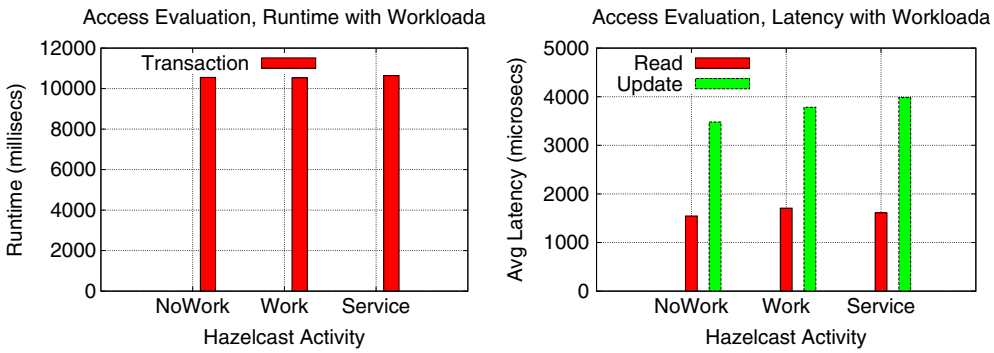

FIGURE 9 Map access evaluation (runtime and latency)

- Work - This case has some additional activity (see Section 2.5.1) going on in addition to serving the requests of YCSB clients such as map read/write/delete, etc. However, there is no service layer; this implies the absence of KeyValueServe. The activity is performed by the Hazelcast cluster directly. (Java APIs)
- Service - This refers to the presence of KeyValueServe performing the activities. The service performs the required activity using Hazelcast APIs in addition to serving the client requests (Python, Java API).

To summarize, the task of serving client requests is common in all the cases. The first "NoWork" case has no additional activity going on, the second "Work" case does some additional work (see Section 2.5.1) and carries it out *directly* but without the service layer, and the third "Service" case does the same work (see Section 2.5.1 for fair comparison) *indirectly*. It uses the Service layer to perform the activity via the Hazelcast APIs. This additional activity is described below in Section 2.5.1.

2.5.1 | Fair comparison

Three cases of activity performed by the cluster or service have been considered in the experiments:

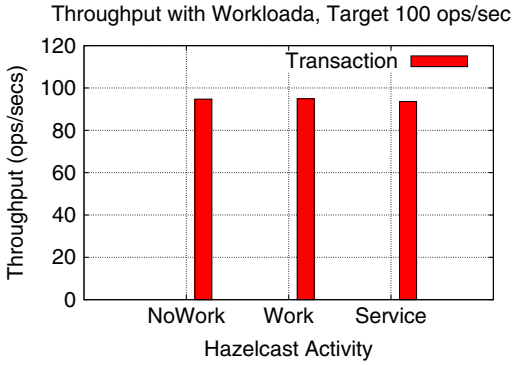

FIGURE 10 Map access evaluation (throughput)

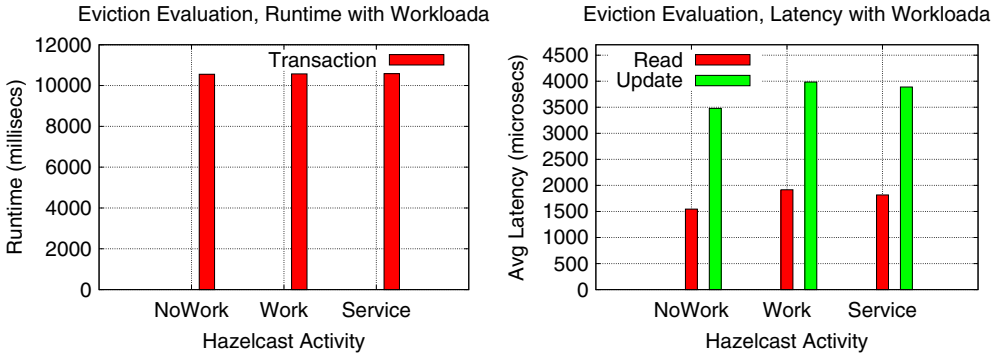

FIGURE 11 Map eviction evaluation (runtime and latency)

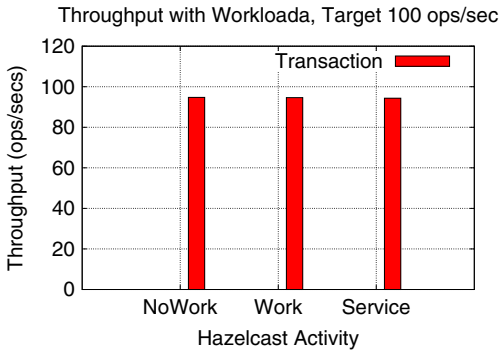

FIGURE 12 Map eviction (throughput)

- *Scalability*: By scalability, we refer to the feature of controlled scalability as mentioned in Section 2.3. The service can either spawn new cluster instances or shut them down dynamically for consolidation based on the cluster load. In this case, experiments were conducted to see the overhead of starting new instances. The objective was to check if those additional instances created during the experimental run caused any significant change in the performance. The operation being performed here is instantiating new instances. The client requests are served by default.
- *Map Access*: In addition to serving the client requests, the distributed map is being accessed for updates. Read operations are performed during experiments.
- *Map Eviction*: In addition to serving the client requests, the map entry eviction is performed by the service. Deletion operations are performed during experiments.

The goal was to evaluate perceived response at the client end with additional activity or service-related functionality happening in parallel on the cluster. We wanted to check the impact on the performance both with and without the service layer and with and without any additional cluster or service activity. The service layer is tightly coupled with the key-value store and leverages its exposed APIs. From Figures 8, 10, and 12, we observe that near target throughput is achieved for all the three activities. For a target of 100 ops/sec, we procure around 95 ops/sec. There is a slight reduction in the throughput from the target because of the locking and unlocking operations on data structures. Moreover, for each activity,

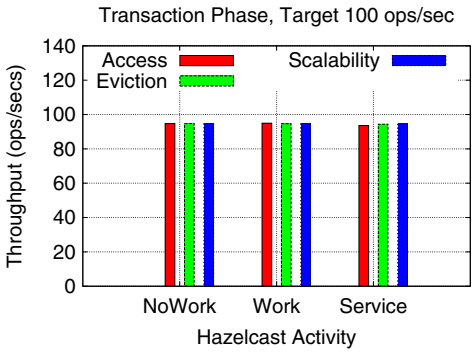

FIGURE 13 Transaction phase throughput

across the three cases of NoWork, Work, and Service, the disparity is not much indicating that the service layer is not impeding the performance. From Figures 7, 9 , and 11, we observe that the overall latency increases slightly with a negligible drop in throughput for the *Work* and *Service* cases, which because of the distributed map operations being done as opposed to the naive case *NoWork* when no operation is performed. Figure 13 clearly shows the transaction phase throughput, where the perceived throughput is seen close to 100 ops/sec for all the three cases. There is no overhead imposed by the service layer in particular since we do not see any performance degradation from *Work* and *Service* cases. Thus, the KeyValueServe service layer is not contributing to any performance degradation.

2.6 | Discussion

The design considerations of multi-tenancy, persistence, controlled scalability, and flexible cost model are the core strengths of *KeyValueServe*. Such a service model is apt for high-volume data-intensive computations with performance guarantees. Both efficient resource multiplexing as well as quality of service can be achieved through such a service design.

2.6.1 | Implications of disk-based access

Our work focuses on in-memory data grids since memory is frequently a bottleneck in low-latency fast-access stores. The premise of this paper is the cached data in RAM. Hence, we propose ideas of garbage collection to store recent and frequently accessed data. Key-value stores offer APIs to support disk-based access. Disk-based access is one possibility, but it will increase the response time and definitely lower the overall throughput. Instead, we utilize a multi-tenant performance *optimization* that depends on channel reuse (see Section 3.7) and is orthogonal to RAM and disk-based access conflicts. Our customized garbage collection feature will function correctly even if most of the keys are not cached and tenant's requests are mostly fetched from the disk. Certain other features such as the cost model and persistence will remain unaffected. It will be difficult to ensure fine-grained disk access since concurrent accesses are more expensive for disks. It should be noted that unlike the prior works such Argus[34] and Libra,[35] KeyValueServe does not focus on disk-I/O cost models or deal with disk or cache reservations and scheduling.

2.6.2 | Extension to other key-value stores

We stress that KeyValueServe is *key-value store agnostic*, ie, it is not Hazelcast specific. However, KeyValueServe is not independent of the key-value store used. It definitely needs some key-value store. In other words, the service is compatible with most contemporary key-value stores such as Memcached, Redis, etc. Ideas of controlled scalability, fined-grained multi-tenancy, persistence model, etc can certainly be adapted on other key-value stores, provided that they are compatible in terms of having a decentralized, peer-to-peer model. Today, most key-value stores, if not all, fulfill this requirement. Needless to mention, necessary customizations and implementation specific tailoring is required based on which key-value store is used.

3 | CONTENTION DETECTION AND PERFORMANCE OPTIMIZATION

In this section, Hazelcast is evaluated to determine and measure the presence of contention leading to performance degradation with increasing numbers of clients. This section illustrates a drop in the per-client throughput and presents a novel way to optimize performance in the presence of increasing numbers of clients.

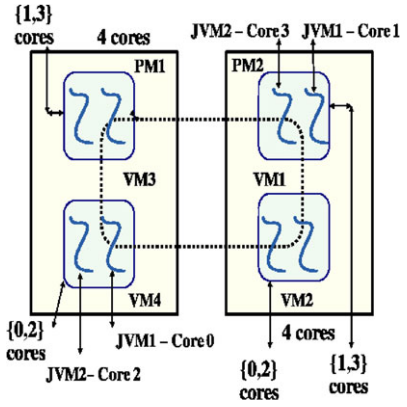

FIGURE 14 Experimental set-up

3.1 | Evaluation methodology

Experiments were conducted on a local cluster, where each cluster node is equipped with a quad-core Xeon 2.53 GHz CPU and 8-GB of memory connected to a Gigabit network switch. Each host ran Ubuntu 12.04 64-bit and RedHat 64-bit with KVM 0.9.8 and KVM 0.10.0, respectively. The guest VMs run Ubuntu 12.04 32-bit and are configured with two virtual CPUs and 4-GB of memory. An 8-instance Hazelcast cluster is set up across 2 hosts and 4 VMs as shown in Figure 14. *JVM* and *Hazelcast server instance* are used interchangeably to indicate a Hazelcast cluster instance henceforth. A separate host outside the cluster ran multiple instances of the YCSB[32] client to create multi-tenant workloads. The client host did not run any other application and is not a bottleneck in any of the experiments since it did not interfere with the cluster resources. The observed performance is described in terms of the overall throughput in *ops/sec*.

3.2 | Performance degradation

To understand client performance, experiments are conducted on the 8-instance cluster set-up. Figures 15 and 16 clearly indicate that the throughput decreases and latencies increase with more clients. This is expected as the overall cluster load increases and the system saturates after a point. Investigations regarding what causes this performance drop were carried out across two major directions: a) JVM-VCPU pinning (Section 3.3) to check whether context switches or migrations of threads across cores affect performance and b) Multiplexing (Section 3.7) client channels to reduce the number of runnable instances per client. These are described below.

3.3 | JVM-VCPU pinning

This section observes the effects of pinning a JVM to a specific physical core. The idea is to prevent thread migration across cores, to increase cache locality, and to reduce the overall context switch overhead, which might arise due to *contention*. Figure 14 depicts a two-level pinning, where every JVM has access to two physical cores. A symmetric set-up was deliberately designed to avoid any unnecessary bias in resource allocation for a JVM.

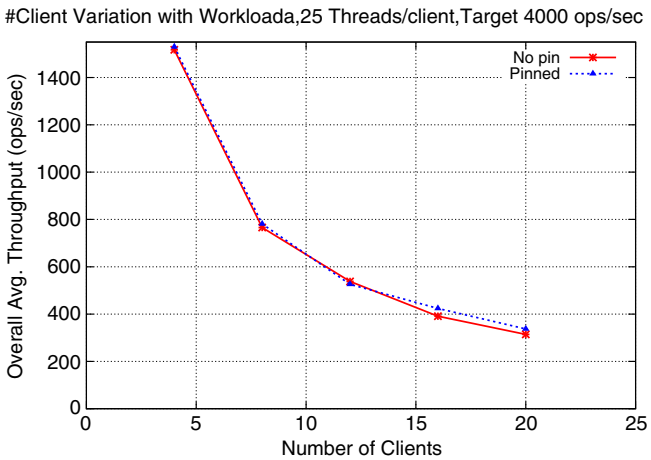

FIGURE 15 Performance drop

FIGURE 16 JVM-VCPU pinned versus unpinned

Eight JVMs were running on 4 VMs, two on each. Each VM was assigned 2 vCPUs. Each vCPU was pinned to 2 physical cores. Each JVM was pinned to 1 vCPU. *taskset*, a Linux utility that internally uses *sched_setaffinity*, was used for pinning, along with changes in the VM configuration file. The number of clients was varied from 4 to 20.

Two sets of experiments were conducted: a) every YCSB client was started with 25 threads and b) the overall thread count in the system from the client's perspective remained fixed. Clients were distributed equally among the 8 server instances to avoid unevenness. In the former case, with an increase in clients, there is an increase in the total number of threads used to create the workload (4 clients, 25 threads each, for a total of 100 threads, 8 clients, 25 threads each, for a total of 200 threads). In the latter case, irrespective of the number of clients, the total number of threads is fixed at 200. In other words, for the latter case, the number of threads per client was varied to generate the workload based on the number of clients. This thread count refers to the *number of client threads*. The overall number of system-level threads created by the Hazelcast cluster remained fixed at all times from the server's perspective. The goal was to check if the client thread count had any significant impact on performance. In the load phase, `Workloada` was used consisting of 50% gets and 50% puts. In the transaction phase, a *Zipfian* distribution was used setting the *target* (expected per client throughput) to 4000 ops/sec for all experimental runs.

As seen in Figure 16, there is no change in the performance with pinning. Even a thread count variation does not affect the performance. Hence, thread migration context switches do not contribute *significantly* to the overhead. Figure 17 shows an increase in the update latency with an increase in the number of clients with a *larger* thread count. However, the average update latency perceived by each client does not deviate much if the overall number of threads generated by the workload remains fixed. The same trend is observed in Figure 18 for the 95th percentile latency illustrating the fact that 95% of the operations completed within the indicated latency on the Y-axis. This indicates that *when the workload is well distributed across the clients by keeping the overall client thread count constant, there is less variation of the average response time.* Every client connects to one instance and delegates its requests to that member. Increasing the number of threads per client further increases parallelization, which increases the overall latency. Figures 16, 17, and 18 clearly indicate that *pinning does not help* in mitigating contention effects. However, a well-distributed workload across multiple clients improves the perceived response time.

Figure 19 illustrates that when the demand (*target throughput per client*) is doubled, the performance does not degrade further, which means that the system is already serving at its full capacity. Demanding more will not improve the per-client throughput. Once the system reaches saturation, increasing demands do not affect throughput; no further degradation is seen.

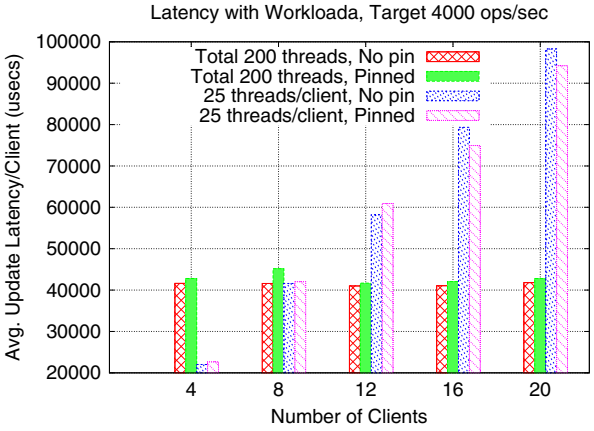

FIGURE 17 Update latency pinned versus unpinned

FIGURE 18 95th percentile latency plot

FIGURE 19 Client variation with different target throughput

3.4 | System throughput

This section discusses the performance behavior of the overall server system. Figure 20 illustrates that increasing the number of clients degrades the per-client throughput. The overall system throughput considering all clients from the cluster's perspective never exceeds 6300 ops/sec. When using 25 threads/client, which is our system threshold, the aggregate throughput of all clients reaches as high as 6300 ops/sec as shown in Figure 21.

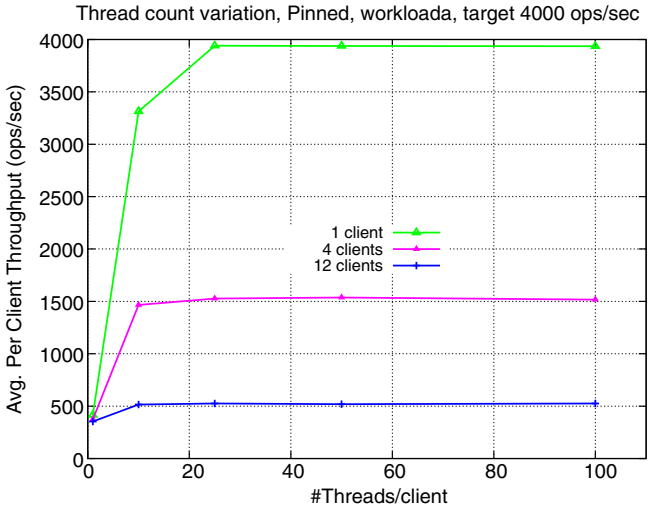

FIGURE 20 Client side thread variation

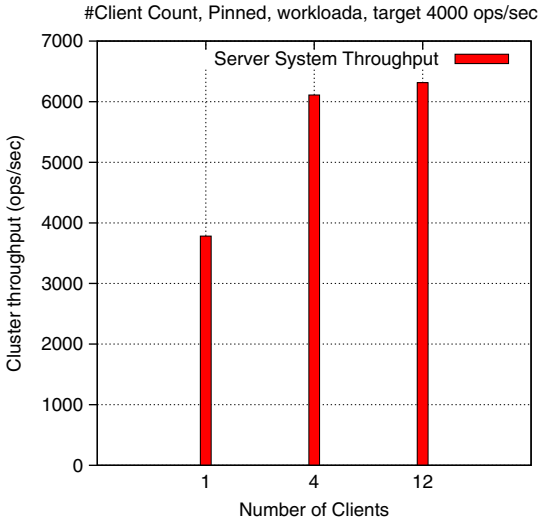

FIGURE 21 Cluster throughput

Considering the limited scale of our experiments, beyond 25 threads/client, there is no performance improvement. With increasing clients, contention increases and the per-client throughput decreases. However, increasing the number of client side threads *does not* degrade performance.

3.5 | Insert order and distribution type

The YCSB load phase uses hashed inserts by default where keys get hashed to specific slots in the database. In case of ordered inserts, the keys get inserted based on the order of keys. Experiments were conducted with ordered inserts and uniform distribution to see the way keys are inserted

FIGURE 22 Hashed and Zipfian versus ordered and uniform distribution

FIGURE 23 Hashed and Zipfian versus ordered and uniform distribution

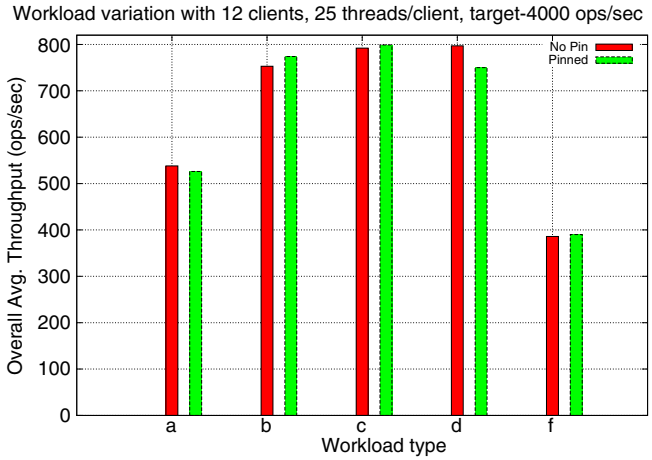

FIGURE 24 Workload type variation

and retrieved impacts performance. Intuitively, it depends on the Hazelcast instance location where the keys get stored on the basis of the hashing algorithm (see Section 2.1.)

Figure 22 shows that hashed inserts are as good as ordered inserts. We do not see any tangible performance variation whether the transaction phase follows a Zipfian or uniform distribution in Figure 23. This implies that *if all the clients have a similar transaction pattern, concurrent data accesses invariably cause contention no matter where the data are stored*.

3.6 | Workload-type variation

Most experiments were conducted with `Workloada` consisting of 50% gets and 50% puts. Two experiments were conducted with other workload types to observe characteristic behavior. While `workloadc` is read only, `workloadb` has 95% reads and 5% updates. `Workloadd` with 95% reads and 5% inserts, and `Workloadf` with 50% reads and 50% *read-modify-writes* were also used in one experiment. Figure 24 confirms that pinning has no impact on the throughput for different workload types. The larger the fraction of updates, the lower the throughput is. However, *read-modify-writes* in `Workloadf` are costlier than updates. Figure 25 shows that updates take more than twice the time than reads.

Atikoglu et al[22] mentioned that reads are more popular than writes in such stores. Even the read throughput (shared lock over exclusive lock used in writes/updates) is considerably throttled due to contention. Improving the operational latency is indeed important for a better tenant service.

3.7 | Multiplexing client channels

We successfully detect contention and throughput degradation with the increasing numbers of clients in Section 3.2. We also found that thread migrations and context switches are not causing the performance drop in Section 3.3. For multi-tenancy, concurrent connections use the network more aggressively. This motivates our next questions, ie, if we reduce the number of parallel simultaneous connections and feed the requests with

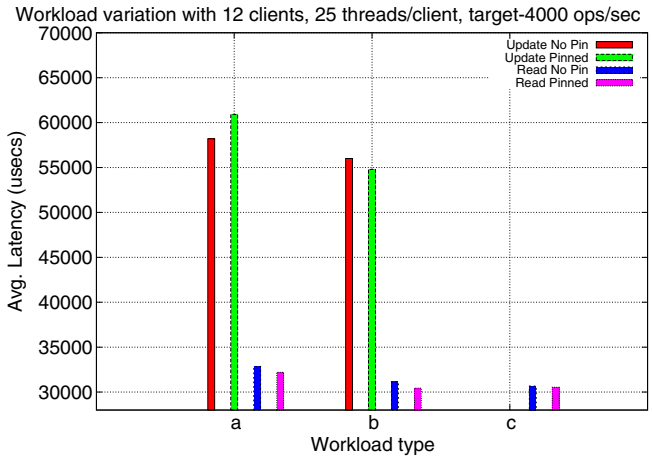

FIGURE 25 Latency variation with diverse workloads

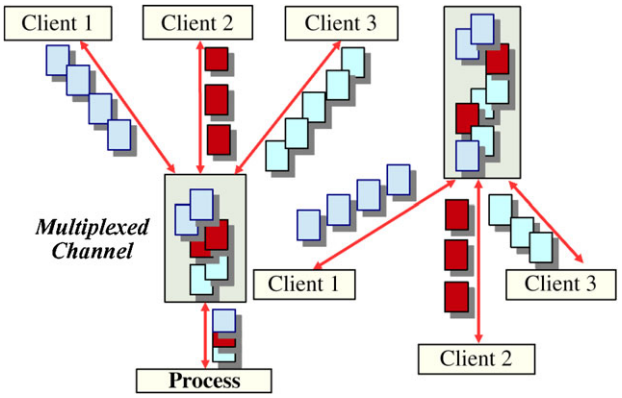

FIGURE 26 Multiplexing channels in Hazelcast

fewer channels will the per-client throughput decrease the same way or will it improve? Is it I/O contention? How should we handle I/O to reduce contention? This inspired the idea of ensemble data stream processing that we shall discuss in this section.

We describe our observation with modifications to Hazelcast. We studied and then modified the *Java* source code in an effort to pipeline multiple client requests through a single proxy entry point. Our observations indicate a performance improvement with the increasing number of clients. Prior experiments confirmed that keeping the number of system-level threads created by the Hazelcast cluster fixed, while increasing clients invariably degrades the throughput. We wanted to see if the performance is affected by reducing the number of concurrent active connections through multiplexing multiple client connections. This reduces the number of runnable I/O threads needed to handle the client threads. Our study indicates a performance improvement over non-multiplexed connections.

As mentioned in Section 2.1, every time a client connects to the Hazelcast server instance, the socket acceptor thread establishes a connection. Data is read from the client channels and processed for further read or write operations through different handlers. When requests of multiple clients are coalesced into a single channel before processing the read or write requests, there is a small initial delay to start with. The idea is to form an ensemble of "n" data streams forming a long-lived connection rather than multiple independent short-lived data streams. Multiple concurrent connections exacerbate network congestion compared with a single multiplexed channel, resulting in contention. Every YCSB client generates requests in close temporal proximity, which further increases the number of connections at the Hazelcast cluster, thereby increasing the overhead. Using our optimized approach, as the number of clients increases, the number of runnable instances created decreases substantially, with the coalesced channel not only making up for the initial time lag but improving the overall throughput as seen in Figure 27. A single long-lived connection is less aggressive in the limited network bandwidth compared to multiple parallel connections, weakening the adverse effects of network congestion.

Using fixed-size thread pools to limit additional thread creation for task execution is a well-known technique. Hazelcast can be configured with a fixed-size thread pool for the purpose of a distributed task execution. However, it should be noted that such fixed-sized thread pools aim to reduce the excessive thread creation overhead during voluminous task execution. They do not aid in reducing the overhead of maintaining multiple active client channels right after the clients establish connections with the cluster. In all our experiments, the default thread pool size is configured and used. Our approach tries to reduce the overall number of simultaneous active connections created when the number of clients increases, restricting the amount of congestion in the system. This should not be confused with the internal executor's threads or queues, which can be limited with the thread pool size configuration. In addition to this, in certain situations, leveraging the benefits of required thread pool size for better performance may not be always possible. Results imply that the throughput is well correlated to the number of active client connection instances (threads); a lower number of connections indicates better performance. This observation conforms to the claims made in the TCP analysis work by Balakrishnan et al.[36]

The `netty`[37] library was used to multiplex multiple client connections with changes in Hazelcast's `nio`-based connection implementation. The contents of multiple registered clients (inbound channels) are multiplexed to the contents of a single outbound channel containing the requests of multiple different clients. This single outbound channel interacting with the `nio` selector reduces the overall I/O because unlike before, the selector does not need to select between multiple channels anymore. This improves the overall balance per cluster instance. We noticed that Hazelcast created fewer threads (I/O, service, execution) than the naive implementation owing to the reduced outbound channel. Unlike before, the selector does not need to poll between multiple channels reducing the overall imbalance per instance. Figure 26 shows the modifications made by passing the contents of multiple socket channels into a single channel before processing it. Each experiment was repeated 3 times, and both mean and standard deviation are reported.

Figure 27 shows the improved performance with multiplexed connections with as many as 8 clients. As the number of clients increases, pipelining overhead increases and there arises a problem with buffer allocation and writing on the outbound channel. However, a marginal improvement in the *per-client throughput* justifies our claim that, indeed, multiple connection instances started for every client right at the outset cause contention, even though the internal data structures used are *asynchronous and non-blocking*. These parallel connections, spawning multiple additional threads

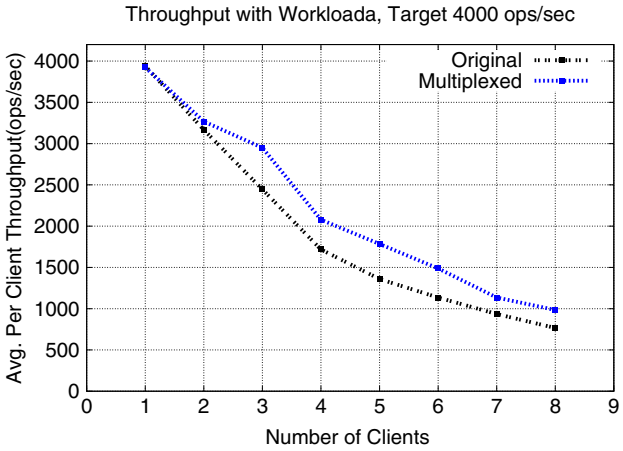

FIGURE 27 Multiplexed connections

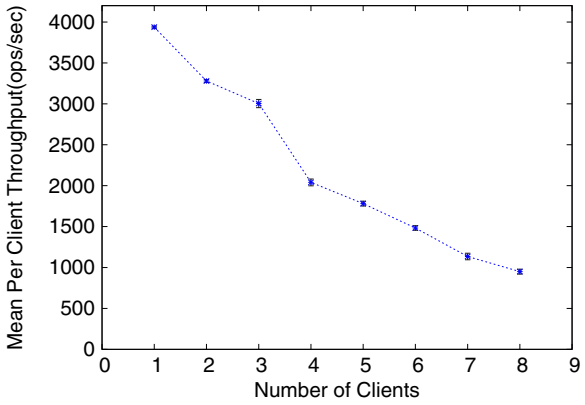

FIGURE 28 Throughput mean and standard deviation

along the way, hurt the overall performance. Figure 28 shows the mean and standard deviation of the experiments. The standard deviation did not exceed 50, and the percentage increase in the throughput is more than 10% for certain cases (see Figure 27).

Some experiments were conducted by over-stressing the system over a limited scale. This is a limitation considering the fact that demanding much higher than the system's maximum threshold (which is a possibility in a real-life scenario) may not be appropriate for performance optimization evaluations. Figure 27 confirms that extrapolating the idea of limiting entry points to a cluster is a viable method to deal with contention problems compared with other solutions of changing the partitioning or scheduling schemes in data stores based on the tenant demand.

There are no side effects of this approach. However, there always exist buffer size limitations. Hence, there will always be a bound on the number of outbound channels required to multiplex a specific number of inbound channels. Thus, the appropriate selection of n (number of client connections) and m (number of channels) is required, through experimentations in a specific configuration. We successfully demonstrate the idea that the per-client throughput improves if the number of parallel concurrent connections is decreased. This holds true for higher numbers of clients as well. However, what m (multiplexed channels) fits what n (incoming client requests) depends on system buffer size, request size, the maximum acceptable system load, network bandwidth, etc. Making this technique scalable is part of our future work. Nevertheless, our results indicate that the abstraction of ensemble processing can be useful to derive a statistical multiplexing scheme by which n clients can be processed through m channels conforming to buffer size limitations.

4 | RELATED WORK

There has been considerable recent research on storage services, performance isolation, and resource management. In this section, we discuss how our work distinguishes itself from the prior state-of-the-art services.

Existing services such as S3 and DynamoDb[28,30] from Amazon and Cloudant[29] from IBM do not exhibit multi-tenancy in the truest sense at par with our definition (see Section 1.2.1). Clients use separate containers or virtual machines for their work without really sharing resources at a

finer granularity, which KeyValueServe provides. Our work discusses how tenant authentication with controlled access can aid multi-tenant sharing across distributed data structures and cluster instances, distinguishing KeyValueServe from the existing storage services.

Pisces[18] enforces system-wide fair sharing and high resource utilization by performing partition placement, weight allocation, replica selection, and fair queuing. Membase key-value store is evaluated. They partition based on demands, assign local weights based on global sharing, choose replica in a weight sensitive manner, and prioritize dominant resource sharing to guarantee fairness. It is based on a centralized controller and requires huge modification of the key-value store, which is unwanted for a suitable cloud service. KeyValueServe supports multi-tenancy at the middleware level, which is less costly than the infrastructure level. Google's Borg[38] is a centralized scheduler, which uses priority-based round-robin scheduling through feasibility check and scoring. It has a master-slave structure where the Borgmaster polls Borglets offering good scale and performance through replicas, cached copies, and stateless link shards. Pisces and Borg[18,38] are intrusive centralized schedulers focusing on resource allocation and management, complementing our work focusing on contention analysis.

MROrchestrator[39] is a resource allocator (typically for Hadoop map reduce jobs) agnostic of interference. Delay scheduling[40] has been used on HDFS systems. But how HFS (Hadoop's Fair scheduler) alone will work for data grids remains an open question. These centralized solutions do not aim at fine-grained multi-tenant data store services.

Cake[41] proposes two-level schedulers using HBase and HDFS to provide differentiated scheduling. They chunk large requests, provide different queues for batch and front end requests, and enforce allocations based on (service level objective) SLO-compliance and queue occupancy. Mercury[42] proposes a hybrid resource management framework that supports the full spectrum of scheduling from centralized to distributed. They regulate the knobs of execution guarantees and scheduling overhead by offloading work from centralized scheduler to auxiliary set of fast schedulers making distributed decisions extending the Hadoop YARN framework. Sparrow[43] proposes a stateless decentralized scheduler, which achieves improved performance through batch sampling and late-binding strategies. Omega[44] designs a distributed multi-level scheduler focusing on scalability without any emphasis on the multi-tenant performance. Mesos[45] and YARN[46] propose two-level schedulers with offer and request-based resource managers. Apollo[47] aims at highly utilized load balanced clusters. Hawk[48] proposes a hybrid scheduler for long and short jobs to leverage the advantages of both centralized and distributed schedulers. Hawk performs better than Sparrow[43] for short jobs under high load by leveraging the idea of work stealing. Several research efforts[41-45,47,48] either propose a comprehensive scheduler infrastructure, which are complex and unfit for our target systems or are not fine-grained enough focusing on Hadoop-style applications. Such decentralized solutions do not focus on ensuring consistent tenant performance such as ours.

SmartSLA[49] evaluates popular machine learning techniques such as linear regression and boosting to optimize resource allocation in an intelligent fashion. KeyValueServe does not focus on infrastructure cost or statistical analysis but on the coalescing of I/O requests for better multi-tenant performance.

GD-Wheel[24] and CAMP[23] propose new key replacement strategies in caches to have high cache hits based on the recency or frequency and re-computation costs, completely orthogonal to our objectives. MBal,[20] CloudScale,[50] and SCADS[51] perform cluster load balancing mitigating hot-spots through key replication or data migration differing from our work. Argon,[52] Walraven et al,[53] EyeQ,[54] SQLVM,[55] Pulsar,[21] Das et al,[56] IOFlow,[57] Zeng and Plale,[58] and PriDyn[59] focus on fairness, either at a coarser granularity dealing with virtual machines, network isolation, higher-level abstraction, or centralized quantum-based scheduling in contrast to our work. Anderson et al[60] quantified key-value store consistency through evaluation. Atikoglu et al[22] unveiled workload insights of real-life traces, which can have implications on the cache configuration. These give useful hints about patterns of real-life usage to analyze the impact of their performance under load and scale.

Cao et al[61] assessed the key-value store performance analysis too, but they used Memcached and Redis unlike Hazelcast. They focus only on the resource overhead unlike our proposal of service modeling and performance optimization schemes. Das et al[62] proposes the channel reuse–based performance optimization technique to improve client performance. KeyValueServe extends the earlier work by providing a multi-tenant cloud service model with detailed experimental evaluation pertaining to service and performance. MeT[63] proposes a cloud-enabled framework to reconfigure a cluster for dynamic workload changes. MeT takes the homogeneity and heterogeneity of nodes into account and proposes a decision maker, which distributes data partitions based on classification and node grouping. While KeyValueServe's controlled scalability adjusts instances based on the cluster load, the partitioning logic is unaltered. Different stores have unique data placement and redistribution policies; we do not want to tamper with data partitioning logic to make our model generic. KeyValueServe, unlike MeT, proposes a multiplexed channel reuse–based scheme to improve the throughput along with several design features, without considering node classification and group data assignments. Argus[34] proposes a workload-aware resource reservation NoSQL store. Argus targets system-wide fair share like Pisces[18] and uses stochastic hill climbing to find optimal resource reservations with changing workload resource demands. Unlike Argus and Pisces, KeyValueServe does not perform elastic resource reservation but combines multiple client requests to alleviate network contention while improving throughput. Libra[35] proposes an I/O scheduling framework for handling I/O interference by charging an I/O operation proportional to its actual resource usage. Unlike Libra, KeyValueServe schedules I/O requests before getting to the persistence engine. Libra is situated below the persistence engine to schedule I/O in a deficit round robin fashion. KeyValueServe leverages the idea of adjusting I/O requests based on ensemble I/O through fewer outgoing channels instead of weighted fair-share approaches. A-Cache[64] resolves cache interference for multiple workloads by tracking cache re-use ratios of different workloads. A-Cache uses cache throttling to improve throughput while KeyValueServe improves the throughput by leveraging the strength of com-

bined client requests over multiple parallel requests. The fundamental ideas are different; moreover, KeyValueServe discusses a service model with features aiding multi-tenant performance.

Existing database and key-value store services perform sharing at a much coarser granularity, leaving scopes of resources under-utilization and resulting in lower degrees of concurrent accesses. Agrawal et al[27] discussed several forms of multi-tenancy in databases as explained in Section 1.2.1. Directing research efforts in enriching the functionality supported by key-value stores and designing scalable, elastic and autonomic multi-tenant systems are emphasized. We have taken a step in that direction. This paper discusses KeyValueServe (Section 2.3), a service that enables multi-tenancy by governing access privileges at the level of data structures, cluster instances, and VMs. Prior work has investigated performance of distributed storage systems, but focusing on sources of contention and ensuring consistent client response still remains challenging. The existence of performance degradation was demonstrated (Figure 16), and a solution (Section 3.7) was shown to especially aid cloud computing infrastructures.

5 | CONCLUSIONS AND FUTURE WORK

In this paper, we propose KeyValueServe, a low-overhead service with its major models of multi-tenancy, persistence, and cost. The Hazelcast in-memory data grid was evaluated in the context of multi-tenancy to assess its performance. Our findings unveil the following interesting insights: JVM-VCPU pinning does not help; maintaining parallel independent client connections with their own thread data structures and work queues degrades the throughput. Even though the internal data structures used are *asynchronous and non-blocking* intended for sharing, an increase in end-to-end separate connection instances with increasing clients causes resource contention. Our results indicate that an ensemble of multiple client connections together forming a single coalesced data stream can alleviate contention.

Future work aims to look at *novel performance optimization* techniques and overlay service delegation protocols to make KeyValueServe more resilient and performance efficient. Consideration of real cloud deployment and a thorough QoS evaluation of KeyValueServe on a larger scale with complex workloads are planned in the future. This performance study also provides a new perspective of looking at the contention problem. The best way to process data through a reduced number of data streams should be analyzed further. Another area for further investigation is comparing and integrating techniques using thread pools and optimized thread pool sizes with our approach of using a combined channel for requests from multiple clients.

ACKNOWLEDGMENTS

The authors thank Professor Xiaohui (Helen) Gu for her support and insightful suggestions in the initial stages of this work. Part of this work was done while Anwesha Das was a summer intern at IBM Research. This work was supported in part by NSF grants 1217748 and 0958311.

ORCID

Anwesha Das http://orcid.org/0000-0003-4441-6449

REFERENCES

1. Chang F, Dean J, Ghemawat S, et al. Bigtable: a distributed storage system for structured data. *ACM Trans Comput Syst*. June 2008;26(2). http://doi.acm.org/10.1145/1365815.1365816
2. Cooper BF, Ramakrishnan R, Srivastava U, et al. PNUTS: Yahoo!'s hosted data serving platform. *Proc VLDB Endow*. August 2008;1(2):1277-1288. http://doi.org/10.14778/1454159.1454167
3. DeCandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper Syst Rev*. October 2007;41(6):205-220. http://doi.acm.org/10.1145/1323293.1294281
4. Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop Distributed File System. Paper presented at: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST); 2010; Incline Village, NV. http://doi.org/10.1109/MSST.2010.5496972
5. HBase. 2008. https://hbase.apache.org/
6. Hypertable. 2008. http://hypertable.org/
7. Escriva R, Wong B, Sirer EG. HyperDex: a distributed, searchable key-value store. *SIGCOMM Comput Commun Rev*. August 2012;42(4):25-36. http://doi.acm.org/10.1145/2377677.2377681
8. Sumbaly R, Kreps J, Gao L, Feinberg A, Soman C, Shah S. Serving large-scale batch computed data with project Voldemort. Paper presented at: Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST'12; 2012; Berkeley, CA. http://dl.acm.org/citation.cfm?id=2208461.2208479
9. Geambasu R, Levy AA, Kohno T, Krishnamurthy A, Levy HM. COMET: an active distributed key-value store. Paper presented at: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10; 2010; Vancouver, Canada. http://dl.acm.org/citation.cfm?id=1924943.1924966

10. Menon P, Rabl T, Sadoghi M, Jacobsen HA. Cassandra: an SSD boosted key-value store. Paper presented at: International Conference on Data Engineering; 2014; Chicago, IL. http://10.1109/ICDE.2014.6816732
11. Fitzpatrick B. Distributed caching with Memcached. *Linux J*. August 2004;124:5.
12. Carlson JL. *Redis in Action*. Greenwich, CT: Manning Publications Co; 2013.
13. Lim H, Fan B, Andersen DG, Kaminsky M. SILT: a memory-efficient, high-performance key-value store. Paper presented at: Proceedings of the 23rd ACM Symposium on Operating Systems Principles; 2011; Cascais, Portugal.
14. Joyent Cloud Services. 2004. https://www.joyent.com/
15. Riak NoSQL Solution. 2009. http://docs.basho.com/riak/latest/
16. Cloudant Cloud Service. 2010. https://cloudant.com/
17. CouchDB NoSQL Database. 2005. http://couchdb.apache.org/
18. Shue D, Freedman MJ, Shaikh A. Performance isolation and fairness for multi-tenant cloud storage. Paper presented at: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI); 2012; Hollywood, CA.
19. Nishtala R, Fugal H, Grimm S, et al. Scaling Memcache at Facebook. Paper presented at: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI); 2013; Lombard, IL.
20. Cheng Y, Gupta A, Butt AR. An in-memory object caching framework with adaptive load balancing. Paper presented at: Proceedings of the 10th European Conference on Computer Systems; 2015; Bordeaux, France.
21. Angel S, Ballani H, Karagiannis T, O'Shea G, Thereska E. End-to-end performance isolation through virtual datacenters. Paper presented at: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation; 2014; Broomfield, CO.
22. Atikoglu B, Xu Y, Frachtenberg E, Jiang S, Paleczny M. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Perform Eval Rev*. 2012;40:53-64.
23. Ghandeharizadeh S, Irani S, Lam J, Yap J. CAMP: a cost adaptive multi-queue eviction policy for key-value stores. Paper presented at: Proceedings of the 15th International Middleware Conference; 2014; Bordeaux, France.
24. Li C, Cox AL. GD-Wheel: a cost-aware replacement policy for key-value stores. Paper presented at: Proceedings of the 10th European Conference on Computer Systems; 2015; Bordeaux, France.
25. MongoDB. 2009. https://www.mongodb.com/
26. Amazon DynamoDB. 2012. https://aws.amazon.com/dynamodb/
27. Agrawal D, Das S, El Abbadi A. Big data and cloud computing: current state and future opportunities. Paper presented at: Proceedings of the 14th International Conference on Extending Database Technology; 2011; Uppsala, Sweden.
28. Sivasubramanian S. Amazon DynamoDB: a seamlessly scalable non-relational database service. Paper presented at: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data; 2012; Scottsdale, AZ.
29. Bienko CD, Greenstein M, Holt SE, et al. *IBM Cloudant: database as a Service Advanced Topics*. Armonk, NY: IBM Redbooks; 2015.
30. Palankar MR, Iamnitchi A, Ripeanu M, Garfinkel S. Amazon S3 for science grids: a viable solution? Paper presented at: Proceedings of the 2008 International Workshop on Data-Aware Distributed Computing; 2008; Boston, MA.
31. Johns M. *Getting Started With Hazelcast*. Birmingham, UK: Packt Publishing Ltd; 2015.
32. YCSB. 2010. https://github.com/brianfrankcooper/YCSB/wiki/Getting-Started
33. Ryu KD, Zhang X, Ammons G, et al. RC2–a living lab for cloud computing. Paper presented at: Proceedings of LISA'10: 24th Large Installation System Administration Conference; 2010; San Jose, CA.
34. Zeng J, Plale B. Argus: a multi-tenancy NoSQL store with workload-aware resource reservation. *Parallel Comput*. 2016;58:76-89.
35. Shue D, Freedman MJ. From application requests to virtual IOPs: Provisioned key-value storage with Libra. Paper presented at: Proceedings of the 9th European Conference on Computer Systems; 2014; Amsterdam, Netherlands.
36. Balakrishnan H, Padmanabhan VN, Seshan S, Stemm M, Katz RH. TCP behavior of a busy internet server: analysis and improvements. Paper presented at: INFOCOM'98. Proceedings of the IEEE 17th Annual Joint Conference of the IEEE Computer and Communications Societies; 1998; San Francisco, CA.
37. Netty. 2003. http://netty.io/
38. Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J. Large-scale cluster management at Google with Borg. Paper presented at: European Conference on Computer Systems; 2015; Bordeaux, France.
39. Sharma B, Prabhakar R, Lim SH, Kandemir MT, Das CR. MROrchestrator: a fine-grained resource orchestration framework for MapReduce clusters. Paper presented at: 2012 IEEE 5th International Conference on Cloud Computing (CLOUD); 2012; Honolulu, HI.
40. Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. Paper presented at: Proceedings of the 5th European Conference on Computer systems; 2010; Paris, France.
41. Wang A, Venkataraman S, Alspaugh S, Katz R, Stoica I. Cake: enabling high-level SLOs on shared storage systems. Paper presented at: ACM Symposium on Cloud Computing; 2012; San Jose, CA.
42. Karanasos K, Rao S, Curino C, et al. Mercury: hybrid centralized and distributed scheduling in large shared clusters. Paper presented at: USENIX Annual Technical Conference; 2015; Santa Clara, CA.

43. Ousterhout K, Wendell P, Zaharia M, Stoica I. Sparrow: distributed, low latency scheduling. Paper presented at: Symposium on Operating Systems Principles; 2013; Farminton, PA.
44. Schwarzkopf M, Konwinski A, Abd-El-Malek M, Wilkes J. Omega: flexible, scalable schedulers for large compute clusters. Paper presented at: European Conference on Computer Systems; 2013; Prague, Czech Republic.
45. Hindman B, Konwinski A, Zaharia M, et al. Mesos: a platform for fine-grained resource sharing in the data center. Paper presented at: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI); 2011; Boston, MA.
46. Vavilapalli VK, Murthy AC, Douglas C, et al. Apache Hadoop YARN: yet another resource negotiator. Paper presented at: Symposium on Cloud Computing; 2013; Santa Clara, CA.
47. Boutin E, Ekanayake J, Lin W, et al. Apollo: scalable and coordinated scheduling for cloud-scale computing. Paper presented at: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI); 2014; Broomfield, CO.
48. Delgado P, Dinu F, Kermarrec AM, Zwaenepoel W. Hawk: hybrid datacenter scheduling. Paper presented at: USENIX Annual Technical Conference; 2015; Santa Clara, CA.
49. Xiong P, Chi Y, Zhu S, Moon HJ, Pu C, Hacigümüş H. Intelligent management of virtualized resources for database systems in cloud environment. Paper presented at: IEEE 27th International Conference on Data Engineering (ICDE); 2011; Hannover, Germany.
50. Shen Z, Subbiah S, Gu X, Wilkes J. CloudScale: elastic resource scaling for multi-tenant cloud systems. Paper presented at: Proceedings of the 2nd ACM Symposium on Cloud Computing; 2011; Cascais, Portugal.
51. Trushkowsky B, Bodík P, Fox A, Franklin MJ, Jordan MI, Patterson DA. The SCADS director: scaling a distributed storage system under stringent performance requirements. Paper presented at: Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST); 2011; San Jose, CA.
52. Wachs M, Abd-El-Malek M, Thereska E, Ganger GR. Argon: performance insulation for shared storage servers. Paper presented at: Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST); 2007; San Jose, CA.
53. Walraven S, Monheim T, Truyen E, Joosen W. Towards performance isolation in multi-tenant SaaS applications. Paper presented at: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing; 2012; Montreal, Canada.
54. Jeyakumar V, Alizadeh M, Mazieres D, Prabhakar B, Kim C, Greenberg A. EyeQ: practical network performance isolation at the edge. Paper presented at: The 10th USENIX Symposium on Networked Systems Design and Implementation; 2013; Lombard, IL.
55. Narasayya VR, Das S, Syamala M, Chandramouli B, Chaudhuri S. SQLVM: performance isolation in multi-tenant relational database-as-a-service. Paper presented at: 6th Biennial Conference on Innovative Data Systems Research (CIDR); 2013; Asilomar, CA.
56. Das S, Narasayya VR, Li F, Syamala M. CPU sharing techniques for performance isolation in multi-tenant relational database-as-a-service. *Proc VLDB Endowment*. 2013;7(1):37-48.
57. Thereska E, Ballani H, O'Shea G, et al. IOFlow: a software-defined storage architecture. Paper presented at: Proceedings of the 24th ACM Symposium on Operating Systems Principles; 2013; Farminton, PA.
58. Zeng J, Plale B. Multi-tenant fair share in NoSQL data stores. Paper presented at: IEEE International Conference on Cluster Computing (CLUSTER); 2014; Madrid, Spain.
59. Jain N, Lakshmi J. PriDyn: framework for performance specific QoS in cloud storage. Paper presented at: 2014 IEEE 7th International Conference on Cloud Computing (CLOUD); 2014; Anchorage, AK.
60. Anderson E, Li X, Shah M, Tucek J, Wylie JJ. What consistency does your key-value store actually provide. Paper presented at: Proceedings of the 6th International Conference on Hot Topics in System Dependability; 2010; Vancouver, Canada.
61. Cao W, Sahin S, Liu L, Bao X. Evaluation and analysis of in-memory key-value systems. Paper presented at: Proceedings of the IEEE International Conference on Cloud Computing; 2016; San Francisco, CA.
62. Das A, Mueller F, Gu X, Iyengar A. Performance analysis of a multi-tenant in-memory data grid. Paper presented at: Proceedings of the IEEE International Conference on Cloud Computing; 2016; San Francisco, CA.
63. Cruz F, Maia F, Matos M, Oliveira R, Paulo J, Pereira J, Vilaça R. MeT: workload aware elasticity for NoSQL. Paper presented at: Proceedings of the 8th ACM European Conference on Computer Systems; 2013; Prague, Czech Republic.
64. Ravi B, Amur H, Schwan K. A-Cache: resolving cache interference for distributed storage with mixed workloads. Paper presented at: 2013 IEEE international conference on Cluster computing (CLUSTER); 2013; Indianapolis, IN.

How to cite this article: Das A, Iyengar A, Mueller F. KeyValueServe: Design and performance analysis of a multi-tenant data grid as a cloud service. *Concurrency Computat Pract Exper*. 2018;e4424. https://doi.org/10.1002/cpe.4424