A YCSB Workload for Benchmarking Hotspot Object Behaviour in NoSQL Databases

Casper Claesen, Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen

imec-DistriNet, KU Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
`first.last@cs.kuleuven.be`

Abstract. Many contemporary applications have to deal with unexpected spikes or unforeseen peaks in demand for specific data objects – so-called hotspot objects. For example in social networks, specific media items can go viral quickly and unexpectedly and therefore, properly provisioning for such behavior is not trivial.
NoSQL databases are specifically designed for enhanced scalability, high availability, and elasticity to deal with increasing data volumes. Although existing performance benchmarking systems such as the Yahoo! Cloud Serving Benchmark (YCSB) provide support to test the performance properties of different databases under identical workloads, they lack support for testing how well these databases can cope with the abovementioned unexpected hotspot object behaviour.
To address this shortcoming and fill the research gap, we present the design and implementation of a new YCSB workload that is rooted upon a formal characterization of hotspot-based spikes. The proposed workload implements the Pitman-Yor distribution and is configurable in a number of parameters such as spike probability and data locality. As such, it allows for more extensive experimental validation of database systems.
Our functional validation illustrates how the workload can be used to effectively stress-test different types of databases and we present our comparative results of benchmarking two popular NoSQL databases that are Cassandra and MongoDB in terms of their response to spiked workloads.

Keywords: NoSQL databases · workload spikes · hotspot objects · YCSB workload · performance benchmark · Cassandra · MongoDB

1 Introduction

Context. Many cloud services are running on geographically distributed data centers for offering better reliability and performance guarantees [28]. These cloud services are inherently subject to fluctuations in demand. These fluctuations often are seasonal and thus behave according to predictable patterns (e.g. Christmas shopping patterns are largely similar every year). In such cases, pattern recognition techniques or machine learning algorithms can be employed to determine up front what is the most suited configuration. In the current state of the art, using a wide array of techniques and tactics such as overprovisioning, autoscaling and self-adaptive tuning of servers, fluctuations in service load can be dealt with efficiently, using a combination of reactive and proactive measures.

However, not all fluctuations are equal, and in some cases, the nature of the demand increase can be both explosive and unexpected. One such example is the behaviour of hotspot objects: these are objects in a database that experience a sudden and substantial increase in demand (a.k.a spikes). The canonical example is that of social media items (e.g., a tweet or a video) that have gone viral, but other types of services may also experience similar unexpected peaks or spikes in their respective workload, e.g., emergency service hotlines and information systems during calamities will be faced with similar challenges.

Problem. Existing well-known performance benchmark systems such as the Transaction Processing Performance (TPC) [25] and TPC-C [20] frameworks as well as the Yahoo! Cloud Serving Benchmark (YCSB) [6] are designed to systematically evaluate different storage technologies in terms of how they cope with a number of pre-defined workloads. Although different workloads exist to evaluate how a certain service copes with increasing payloads and fluctuations, there is currently no existing workload in these systems that approximates hotspot behaviour. As such, these existing benchmark systems do not provide us with a clear way to assess the extent to which databases can cope with these types of behavioural patterns and even though a database may be highly scalable and elastic, it may not do so efficiently for the specific case of hotspot workloads.

Contribution. In this paper, we present the design and implementation of a YCSB workload that allow us to systematically benchmark the capabilities of a storage system in dealing with such hotspot-based spikes behaviour. The proposed workload implements the Pitman-Yor distribution [22], which is more effective for stimulating workloads with spikes and allows tuning the parameters of (i) the powerlaw (to set the baseline popularity), (ii) the degree of structuring of the records, (iii) the locality of objects over databases, (iv) the distribution of write, read, update operations, (v) the desired variation in popularity, and (vi) the magnitude and recurrence of spikes.

Validation. We have validated the proposed workload by benchmarking hotspot objects behaviour in NoSQL databases (Cassandra and MongoDB) and comparing the results with the core workloads of the YCSB benchmark. The validation results show that the proposed workload generates (unexpected) spikes, which can be used to assess the ability of databases to cope with peaks in demand and hotspot behavior. In addition, we demonstrate the effectiveness of caching strategies in these databases to meet demand considering unpredictable spikes.

Structure. The remainder of this paper is structured as follows: Section 2 discusses the relevant background and formulates the problem statement for this paper. In Section 3, we present the design and implementation of our proposed workload, which is an extension to the existing workloads supported in YCSB. Then, Section 4 reports the results of functional validation and a more extensive evaluation of two popular NoSQL databases (Cassandra and MongoDB) using our proposed workload. Section 5 provides an overview of the related work, and Section 6 concludes the paper.

2 Background

This section provides the necessary background to keep the paper self-contained. More specifically, Section 2.1 provides a summary of the terminology concerning workload spikes and hotspot objects. In Section 2.2, we discuss the overall architecture of the Yahoo! Cloud Serving Benchmark (YCSB) and also describe different data distribution mechanisms and built-in workloads supported in YCSB. Finally, Section 2.3 formally describes the problem statement of our paper.

2.1 Spikes

There are two types of workload spikes: (i) volume spikes and (ii) data spikes. A volume spike is an unexpected sustained increase in the total volume of the workload, whereas a data spike is a sudden increase in demand for certain objects or in general a marked change in the distribution of popularity of objects. According to these definitions, a data spike should not be a volume spike and vice versa. In practice, however, volume and data spikes often arise simultaneously. Based on the characterizations of Bodik et al. [4], we define *hotspot objects* and *spikes* as follows:

Hotspot objects. An increasing fluctuation of spikes in workloads create hotspot objects. For example, an object is denoted as a *hotspot object* if the change of the workload of this specific object compared to before represents a significant spike. More formally, an object is a *hotspot object* as $\triangle_{i,Ts} > D$ with D a given threshold and $\triangle_{i,Ts}$ the change of the workload of the object i at the starting point of the spike [4].

Workload Spikes. Four factors determine the spikes of a workload: (i) steepness, (ii) magnitude, (iii) duration, and (iv) spatial locality. The steepness expresses how fast the volume of the workload goes up. The magnitude is the difference in popularity between the hotspot objects in the spikes and the normal workload. The duration determines how long the spike lasts. Finally, the spatial locality of a spike defines where the hotspot objects are located. The objects that correspond to spike and more specifically consider all these four factors are called the hotspot objects.

A spike can be represented by a couple of parameters. The symbol s represents a spike. A spike in symbols is then equal to $s = (t0, t1, t2, t3, M, L, N, V)$.

A spike is determined by N the number of hotspot objects and V the variation of the hotspot popularity. M defines the magnitude of the spike. The duration is expressed by $t0$, $t1$, $t2$ and $t3$. The combination of M with $t0$, $t1$, $t2$ and $t3$ determines the steepness. Finally, the L parameter determines the spatial locality of the hotspot objects [4].

2.2 Yahoo Cloud Serving Benchmark (YCSB)

The Yahoo Cloud Serving Benchmark (YCSB) [6] is one of the most popular and frequently-used benchmark systems to evaluate the performance of NoSQL databases. It supports a wide range of NoSQL databases out of the box and is also extensible in this regard. Section 2.2.1 describes the overall architecture of YCSB, while Section 2.2.2 describes different types of workloads currently supported in YCSB.

2.2.1 YSCB architecture. As shown in Fig 1, the architecture of YCSB [6] consists of four core components: (i) the `Workload Executor`, (ii) different `Client` threads, (iii) the `DB interface` layer, and (iv) the `Stats` component. The `Workload Executor` component is responsible for executing specific runs of the benchmark, which involves creating and coordinating a specified number of `Client` threads. These threads execute insert, read, update, delete (CRUD) operations on the target database through the `DB Interface` layer. This layer in turn makes an abstraction of the underlying database in order to support easy switching to different database technologies. The `Stats` component collects all the results (measured latencies) of the experiment.

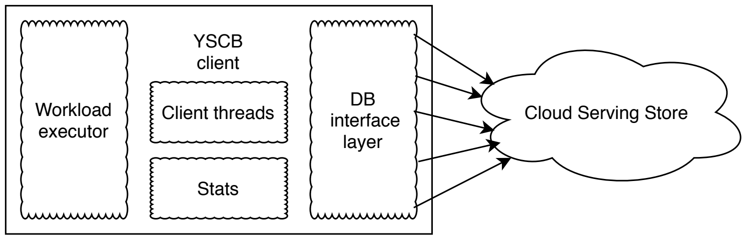


Fig. 1: Architecture of the YCSB benchmark system [6].

2.2.2 YCSB workloads and distribution. Out of the box, YCSB supports three distributions: (i) *Uniform*, (ii) *Zipfian*, and (iii) the *Latest*. In the case of Uniform distribution, all records have an equal chance to occur in the next operation. On the other hand, when Zipfian distribution is chosen, some records are very popular (the head), whereas others are unpopular (the tail). The Latest distribution is the same as Zipfian only the latest added results are set to be the most popular ones. Next to these three distributions, YCSB also supports Multinomial distribution, in which the probability can be configured on a per-item basis. In addition, YCSB currently supports five distinct workloads: *A*, *B*, *C*, *D* and *E*, each with their own characteristics. Table 1 summarizes these workloads, indicating the type of distribution being used and the different types of applications these workloads mimic.

[1] The Zipfian distribution chooses the first key in the range and the Uniform distribution determines the number of records to scan.

Workload	Operations	Record selection	Type of application
A: Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B: Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C: Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D: Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest statuses
E: Short ranges	Scan: 95% Insert: 5%	Zipfian/ Uniform[1]	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

Table 1: An overview of core workloads supported in YCSB [6].

2.3 Problem statement

Existing benchmark systems[2] mainly focus on generating flat workloads and as such lack support to simulate workloads with spikes. These systems do not take into account the sudden change in the popularity of the objects. As an example, some cold (unpopular) objects suddenly can become hot (popular) objects. However, testing how a database deals with sudden and unpredictable spikes is essential to evaluate the resilience and scaling capabilities of the systems. In addition, the current implementations (e.g., YCSB framework) rely extensively on the *Zipfian* (or the derived *Latest*) distribution to determine the popularities of the objects. *These power-law distributions are insufficiently realistic. The popularity of the most popular objects is not high enough or the tail of the distribution falls too slow [4] and as such these distributions can not be used to benchmark hotspot behaviour.* In summary, there is a strong need for: (i) a new workload that supports unpredictable spikes and (ii) a more realistic distribution.

3 YCSB workload for benchmarking hotspot object

In this section, we present our extension of YCSB by introducing a new workload that enables us to benchmark hotspot objects behaviour in different NoSQL databases. As such, a new YCSB workload is introduced that is capable of generating spikes and thus imitating data and volume spikes. The proposed workload uses a new generator that is based on the Pitman-Yor distribution [22]. In addition, extra functionality is added to support a more precise way of configuring the objects in terms of locality and structure.

Fig 2 provides a graphical overview of our proposed architecture. As shown (in bold), we have introduced a number of new components, which include (i) the `SpikesGenerator` component, (ii) the `ObjectDataStore` component, (iii) the

[2] In this paper, we mainly focus on YCSB. However, a more extensive discussion of other benchmark systems is covered in Section 5.

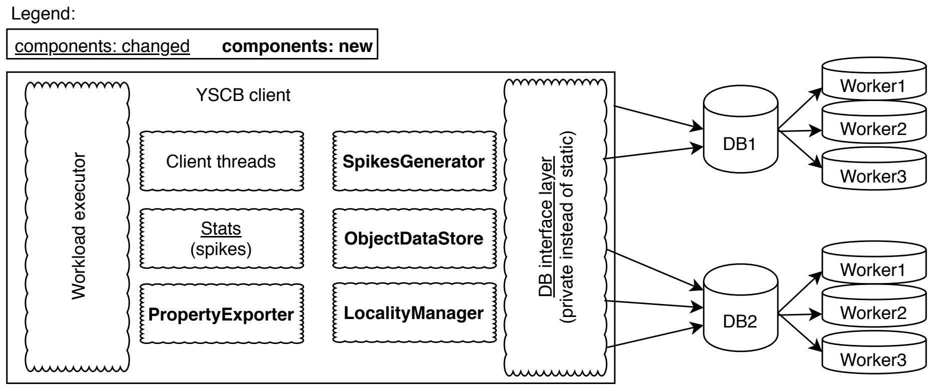


Fig. 2: Architecture of our proposed system, which extends YCSB [6] by introducing a new workload for benchmarking hotspot behaviour in NoSQL databases.

`LocalityManager` component, and (iv) the `PropertyExporter` component. In addition, we have also modified the existing components of the YCSB benchmark. The rest of this section provides a high-level overview of the key features of these new components, which are added in YCSB, while each component is further discussed in detail in the following subsections.

The `SpikesGenerator` component is the key part of our system and is responsible for generating the spikes based on parameters such as the baseline popularity. The `ObjectDataStore` component is an index that keeps information about the intended role and the functionality of each generated object. Examples are the baseline popularity, locality, etc. The `LocalityManager` component provides support for experiments over multiple databases. The `PropertyExporter` component allows exporting these generated properties in order to support performing an experiment with the same overall configuration, and thus increases reproducibility. It also ensures that the information about the records and the generated parameters are the same during the load and the run phase. All parameters have also a forced alternative. For example, the parameters *powerlaw* (1.2) and *maxPop* (0.3) generate the baseline popularity, but the alternative parameter *objForcedPopularity* (0.1, 0.0, 0.2, . . .) sets a fixed baseline popularity.

The YCSB components that were changed to accommodate for this new workload type are (i) the `Stats` component and (ii) the `DB interface` layer. The `Stats` component has been expanded so that more information about the latencies of the hotspot objects can be reported, whereas the `DB interface` layer needed minor changes to allow connections to different databases simultaneously.

The extension maximally adheres to the design principles of the YCSB, extending base classes and leveraging existing configuration facilities where possible. We discuss the introduced components (except for the `PropertyExporter`) in further detail below.

3.1 SpikesGenerator

The `SpikesGenerator` component consists of two classes: (i) *PitmanYorSpikesGenerator* and (ii) *SpikesWorkload*.

PitmanYorSpikesGenerator uses the Pitman-Yor distribution, as explained in the paper of Bodik et al. [4], to determine the baseline popularity (B) of the objects. These popularities are used to decide on which objects the operations execute when no spikes occur. The implementation adds an extra parameter *maxPop* that determines the mapping range from $[0.0, maxPop]$ instead of $[0.0, 1.0]$. At the end, a random permutation is taken from these popularities. The process is called as $PY(1/a, 0.5))$ with a the configurable power-law parameter.

The *PitmanYorSpikesGenerator* is a YCSB generator that extends the existing abstract class *NumberGenerator*. However, a major difference between the new generator and the existing generators is that it records the relevant properties of each object in the *ObjectDataStore*, which is explained in Section 3.2.

To support spikes, a hotspot popularity (H) is calculated with the Dirichlet distribution [4]. The object popularities are set between 0.0 and 1.0, with their sum equal to 1.0. With N the number of hotspot objects and V the variation of the popularity of the hotspot objects (which ranges from 0.0 (equal) to $\frac{N-1}{N^2}$ (heavy tailed)), the parameters for the Dirichlet distribution are calculated as follows [4]:

$$\alpha_i = \frac{N - 1 - V * N^2}{V * N^3}$$

Then after N hotspot popularities are calculated, N objects are chosen with the global locality parameter L to become the hotspot objects. The value of L ranges from 0.0 to 1.0 meaning uniform and heavy-tailed selected over the possible locations. Finally, the popularity P at time t with magnitude factor c_t (0 if normal, $(M-1)/M$ at peak of the spike) is equal to:

$$P_t = (1 - c_t)B + c_t H$$

SpikesWorkload implements a workload class that is capable of simulating spikes[3]. The *SpikesWorkload* class is an extension of the *CoreWorkload* which is responsible for setting the parameters of this class. At pre-determined times, the popularities of an object are switched from the baseline to the spike popularities and the scheduling of the operations is influenced correspondingly. In this way, data and volume spikes can be generated. For this purpose, it uses *PitmanYorSpikesGenerator*. The classes *SpikeObject* and *LocalityObject* are responsible to maintain the information about the spike and the locality on a per-object basis. The *SpikesWorkload* generates the events of the spike in the function *doSpikeEvents*. Possible events are (i) the start ($t0$), (ii) rising ($t0-t1$), (iii) flat ($t1-t2$), (iv) declining ($t2-t3$), and (v) the end ($t3$) of the spike.

[3] In the YCSB config, it will be used when the parameter *workload* is set to $site.ycsb.workloads.SpikesWorkload$.

3.2 ObjectDataStore

The `ObjectDataStore` component represents an index that keeps track of the different properties of each generated object in the *DataObject* record. This record consists of (i) an identifier, (ii) a boolean that indicates if the object is a hotspot, (iii) the baseline popularity, (iv) the hotspot popularity, (v) the locality, and (vi) a boolean to determine if the object is structured or not.

As YCSB supports structured data and not unstructured data, a new parameter *objStructProportion* is introduced. It determines the proportion of the structured items. The value ranges from 0.0 (all unstructured) to 1.0 (all structured). The extra information is stored in the *DataObject* class. When a new record is constructed and the object is unstructured, all fields are concatenated and stored in the first field. In this way, it is possible to simulate blobs.

3.3 LocalityManager

YCSB has been developed under the main assumption that an experiment only involves a single database technology. A setup with multiple databases can be tested through multiple independent experiments on each database. In this way, there is no support to take the data locality aspect of hotspot objects into account. This is necessary to know in detail what happens when spikes occur.

DBContainer solves the problem of supporting multiple databases by retaining the different databases. If the parameter *hosts* is an array of IP-addresses, a container with the corresponding databases is created. For each database operation (insert, read, update, scan, and delete), new functions are implemented with the *DBContainer* as a parameter instead of the *DB*. The database-specific calls are determined at run-time through method overloading, where a specific target database is chosen in the *SpikesWorkload* class by the information of the *DataObject* class of the *ObjectDataStore* component.

As discussed in Section 3.1, the global locality parameter L is used to determine objects that have become the hotspot objects based on their location. The global locality parameter *objLocality* defines if the objects must be grouped on one database (1.0) or spread over different multiple databases (0.0). As such, it determines the locality value where the first IP-address of hosts is mapped to 1, the second to 2, etc.

4 Functional validation

As a functional validation, we illustrate the importance of our proposed workload in terms of benchmarking hotspot objects behavior in NoSQL databases. More precisely, the goal of this functional validation is to show the differences between the core workloads of the YCSB benchmark (cf. Table 1 for more information about different workloads supported in YCSB) and our proposed workload, which is specifically designed to simulate hotspot behaviour in NoSQL databases (cf. Section 3 for more details about our proposed workload). Section 4.1 gives an overview of the experimental setup and also provides details on software and hardware used for the experiments. The subsequent section (Section 4.2) presents the main results with a critical discussion.

4.1 Experiment Setup

In order to validate our approach, we implemented two application prototypes that use different workloads. The first prototype (Prototype YCSB) is based on the core workload of type A (workload_A) of the YCSB benchmark (cf. Table 1), which consists of 50 % read operations and 50 % update operations, generates no spikes, and relies on the Zipfian distribution. The second prototype (Prototype YCSB$^{\text{Hotspot}}$) is based on our proposed workload, which also consists of 50 % read and 50 % update operations, but instead generates spikes and relies on Pitman-Yor distribution. Moreover, in the current implementation, the proposed workload contains one hotspot object (cf. Table2 (a)) for more details about different hotspot object properties).

Table 2(a) describes the popularity and object parameters of our proposed workload, while the spike and experiment parameters are listed in Table 2(b). For both application prototypes, we have used 10 000 records and performed 100 000 operations with the same number of threads (10 threads) where the proportion of read and update operations is 0.5 (50 %). The warm-up phase (first 10 000 operations in our case) is excluded from the presented results, as it involves higher latencies and eventually leads to inconsistency in results.

The experiments are conducted in a client-server environment where the client process runs application prototypes and the server process runs different databases. In our case, both client and server processes run on a single-node setup[4], which consists of Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz x 4 processor and 3.7 GB RAM with Ubuntu Bionic (18.04.4) installed. For all the experiments, we have used both Cassandra and MongoDB with their default configurations: cache is disabled in Cassandra by default, whereas enabled in MongoDB. In addition, we have cleaned both databases at the start of every new experiment.

4.2 Results

This section presents the results of our experiments for both Cassandra and MongoDB databases. Section 4.2.1 presents and discusses the results of the Cassandra database, while Section 4.2.2 outlines the results of the MongoDB database followed by the discussion in Section 4.2.3.

4.2.1 Results of the Cassandra database. The results of all the experiments where application prototypes use the Cassandra database are presented in Fig 3. The left-hand side of the Fig 3 (Figs 3a and 3b) represents the results

[4] The experiments for a multi-node setup will be considered in the future work.

baseline popularity	
powerlaw	1.2
maxPop	0.3
hotspot popularity	
objForced-PopularitySpike	[1.0, 0.0, 0.0, . . .]
all objects	
objLocality	1.0
hotspot objects	
L	1.0
all objects	
recordcount	10000
fieldcount	10
fieldlength-distribution	uniform
fieldlength	35
minfieldlength	25
objStructProportion	1.0

(a) Popularity and object parameters.

one spike	
$t0$	20000
$t1$	26666
$t2$	33333
$t3$	40000
all spikes	
t	50000
M	3
waitingTime-OperationNoSpike	100
characteristics experiment	
operationcount	100000
read-proportion	0.5
update-proportion	0.5
threads	10

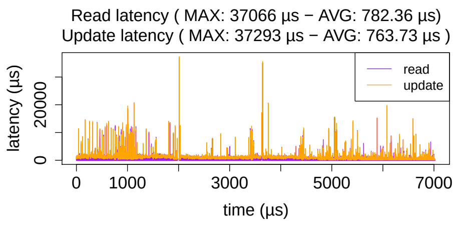
(b) Spike and experiment parameters

Table 2: Different parameters of our proposed workload, which generates spikes and contains one hotspot object.

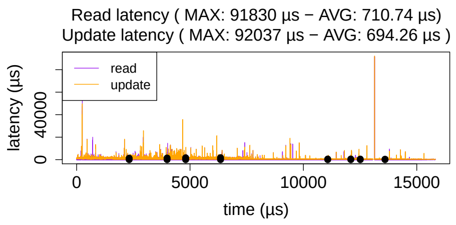
of prototype YCSB, which is based on the core workload of type A (workload_A) of the YCSB benchmark (cf. Table 1), whereas the right-hand side of the Fig 3 (Figs 3c and 3d) shows the results of prototype YCSB$^{\text{Hotspot}}$, which is based on our proposed spike workload.

Figs 3a and 3c display the operation latencies, while Figs 3b and 3d present the number of processed operations per discrete time bucket in terms of throughput. The rainbow visualization of these graphs ensures a better visualization of the bars and does not have any functional meaning. The black dots in Fig 3c and Fig 4c present respectively $t0$, $t1$, $t2$ and $t3$ of the spike as explained in Section 2.1. As visible in the results, Cassandra is optimized for the write-heavy workloads. The maximum latencies of prototype YCSB$^{\text{Hotspot}}$ are much higher when the spikes occur [(91830μs, 92037μs) vs (37066μs, 37293μs)]. The average latencies of prototype YCSB$^{\text{Hotspot}}$ are much lower than prototype YCSB [(711μs, 694μs) vs (782μs, 764μs)]. This goes against intuition, but the small waiting time before each operation to simulate spikes and the relatively small time period of spikes explains it. The comparison of the x-axis of Fig 3a with Fig 3c makes it clear that prototype YCSB requires less time [+/-7000μs vs. +/-15000μs] to execute all operations.

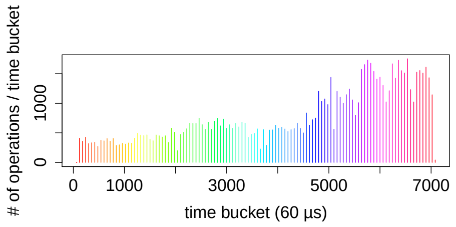
4.2.2 Results of the MongoDB database. The results of all the experiments where application prototypes use the MongoDB database are presented in Fig 4. The left-hand side of the Fig 4 (Figs 4a and 4b) presents the results of Prototype YCSB, which is based on the core workload of type A (workload_A) of YCSB (cf. Table 1), while the right-hand side of the Fig 4 (Figs 4c and 4d)



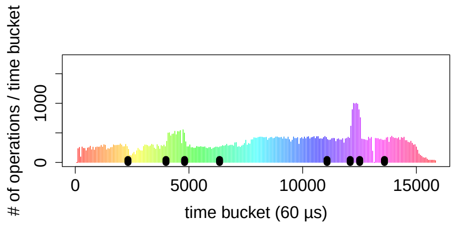
(a) Operation latencies in μs for Prototype YCSB, which is based on the core workload of type A (workload_A) of the YCSB benchmark.



(c) Operation latencies in μs for Prototype YCSB$^{\text{Hotspot}}$, which is based on the proposed spike workload.



(b) Number of processed operations per discrete time bucket of 60 μs (throughput) for Prototype YCSB.



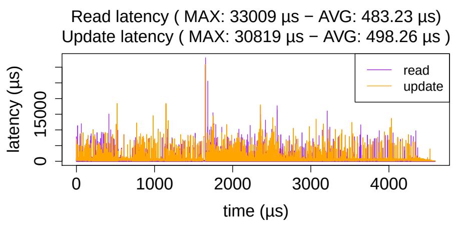
(d) Number of processed operations per discrete time bucket of 60 μs (throughput) for Prototype YCSB$^{\text{Hotspot}}$.

Fig. 3: Results of both application prototypes (prototype YCSB and prototype YCSB$^{\text{Hotspot}}$) which include (i) operation latencies in μs and (ii) number of processed operations per discrete time bucket of 60 μs (throughput) for the Cassandra database.

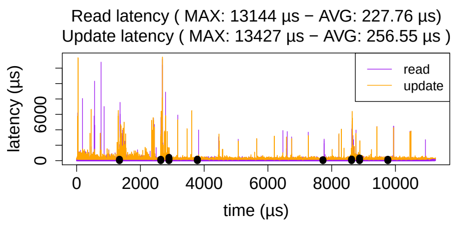
displays the results of Prototype YCSB$^{\text{Hotspot}}$, which is based on our proposed workload. As clearly visible in the results, MongoDB is optimized for the read-heavy workloads. As shown, prototype YCSB$^{\text{Hotspot}}$ again performs better than prototype YCSB [avg:(228μs, 257μs) vs. (483μs, 498μs) and max: (13144μs, 13427μs) vs. (33009μs, 30891μs)]. The reason that prototype YCSB$^{\text{Hotspot}}$ performs better than prototype YCSB is that the new Pitman-Yor distribution (baseline popularity) is, in this case, generating a small amount of very popular items, so that less swap caching occurs. The peaks of the latencies when a spike occurs are clearly visible between the black dots indicating the spikes in Fig 4c.

4.2.3 Discussion. Due to space constraints, this section only focuses on the functional validation of the generation of spikes. Hence, the proof of the functional correctness of the generated popularity, locality and the structure/size of the objects by the corresponding parameters are left out. Figs 3d and 4d clearly show that the workload generates spikes in the throughput that are visible between the black dots indicating the spikes. The corresponding latencies when the spikes occur are also visible in 3c and 4c (indicated with the black dots), and these illustrate the resulting increase in read and write latencies.

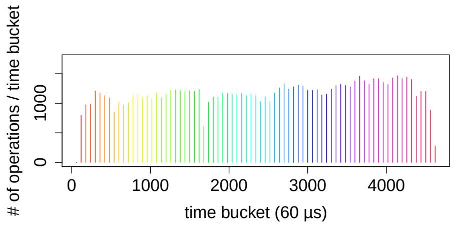
The different results of Cassandra and MongoDB can entirely be attributed to caching: in this case, the results confirm that object caching is a suited tactic



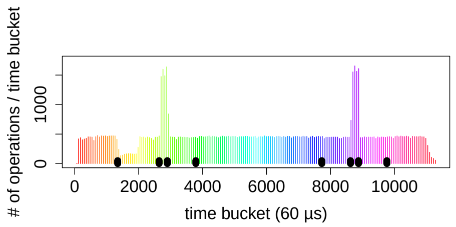
(a) Operation latencies in μs for Prototype YCSB, which is based on the core workload of type A (workload_A) of the YCSB benchmark.



(c) Operation latencies in μs for Prototype YCSB$^{\text{Hotspot}}$, which is based on the proposed spike workload.



(b) Number of processed operations per discrete time bucket of 60 μs (throughput) for Prototype YCSB.



(d) Number of processed operations per discrete time bucket of 60 μs (throughput) for Prototype YCSB$^{\text{Hotspot}}$.

Fig. 4: Results of both application prototypes (prototype YCSB and prototype YCSB$^{\text{Hotspot}}$) which include (i) operation latencies in μs and (ii) number of processed operations per discrete time bucket of 60 μs (throughput) for the MongoDB database.

when spikes occur. As such, we show that the proposed benchmark is an enabler for experimenting with and optimizing different tactics (e.g. caching or adaptive provisioning such as autoscaling) for specific spike-based workload profiles.

5 Related work

Existing benchmarks such as TPC-C [16,20] and TPC-E [26] focus on emulating database applications to compare different relational database management systems (RDBMS). These benchmarks use predefined queries, which are executed within the context of transactions to measure the performance (e.g., throughput) of different RDBMS. Similarly, Difallah et al. [10] proposed an extensible and easy-to-use testbed, which contains fifteen workloads that all differ in complexity and system demands for benchmarking relational databases. Cloud service benchmark such as YCSB [6], on the other hand, is designed to evaluate the performance of distributed databases. Although YCSB is the de-facto standard for evaluating the performance properties of distributed database systems (e.g., NoSQL databases), it fails to adequately mimic the hotspot behaviour in these databases. More specifically, the *Zipfian* (or the derived *Latest*) distributions of YCSB mainly focus on generating flat workloads and as such lack support to simulate workloads with spikes. In addition, these distributions do not take into account the locality of objects and also lack in supporting sudden and unexpected change of popularity in objects.

In recent years, significant research efforts focus on extending YCSB in a number of different ways to support and evaluate other properties of distributed databases. For example, YCSB++ [19] is designed to evaluate non-transactional access to distributed key-value stores. It extends the API to achieve bulk loading of data into databases such as HBase and Accumulo. YCSB+T [9], an extension of YCSB is designed with the ability to wrap multiple database operations into transactions and to improve performance understanding and debugging of advanced features such as ingest speed-up techniques and function shipping filters. Dayarathna et al. [8] conducted an in-depth study focusing on existing benchmarks for graph processing systems, graph database benchmarks, and bigdata benchmarks with graph processing workloads. One such example is XGDBench [7], a benchmarking platform, which is designed to operate in both current cloud service infrastructures and future exascale clouds. XGDBench extends YCSB and mainly tends to focus on benchmarking graph databases (such as AllegroGraph, Fuseki, Neo4j, OrientDB), which are beyond the scope of this work. Kumar et al. [15] proposed a system that extends YCSB in order to enable users to select the right storage system for a given application by evaluating the performance and other tradeoffs such as consistency, latency, and availability.

Barahmand et al. [3] proposed BG, a benchmark system that rates different databases for processing social networking actions using pre-defined SLAs. BG is also inspired by prior benchmark systems such as YCSB and YCSB++ and can be used for multiple purposes such as comparing different databases and quantifying the performance characteristics in the presence of failures. Sidhanta et al. [24] introduced Dyn-YCSB, a system that is built upon YCSB and eliminates the need for users to manually change the workload configurations whenever the workload parameters are changed. According to user-specified functions, Dyn-YCSB automatically varies the parameters in YCSB workloads. BSMA [29] is mainly designed for benchmarking the performance of analytical queries over social media data. In comparison to existing benchmark systems (e.g., YCSB) that only provide a synthetic data generator, BSMA is different in the sense that it also provides a real-life dataset with a built-in synthetic data generator. The real-life dataset contains tweets of 1.6 million users and also allows to generate both social networks and synthetic timelines.

Smartbench [14] evaluates the suitability level of RDBMS in supporting both real-time and analysis queries in Internet of Things (IoT) settings. BigBench [13] is an end-to-end benchmark that contains data model and synthetic data generator to address different aspects (volume, velocity, and variety) of big data. In comparison to previous research efforts that mainly focus on structured data, BigBench also takes into account semi-structured and unstructured data. To accomplish this, the BigBench data model is adopted from the TPC-DS benchmark [18, 23], which is enriched with semi-structured and unstructured data components. BigBench V2 [12] is a major rework of BigBench where a new data model and the generator is proposed that rather reflects simple data models and late binding requirements. In contrast to the previous work, BigBench V2 is completely independent of TPC-DS with a new data model and an overhauled workload. BigFUN [21] is a micro-benchmark that is based on a synthetic social network scenario with a semi-structured data model to evaluate the performance of four representative Big Data management systems (BDMSs): MongoDB, Hive, AsterixDB, and a commercial parallel shared-nothing relational database system. BigDataBench [11] compares the performance of systems for analyzing semi-structured data, including their ability to efficiently process machine learning algorithms in a map/reduce setting. These frameworks focus specifically on evaluating the performance and scalability factors of databases. To cope with the challenge of testing ACID properties, Waudby et al. [27] presented a set of data model-agnostic ACID compliance tests for graph databases.

There also exist other recent works that deal with spikes and variation in workload. Arasu et al. [1] proposed Linear Road, a benchmark to compare the performance of Stream Data Management Systems (SDMS) with relational databases. As such, it simulates a toll system for motor vehicles in a large metropolitan area. This benchmark can mimic a form of spike behaviour that is very use-case specific. Hereby, the solution is not applicable to other use cases. However, our proposed solution can easily be extended and applied to a wide range of use cases. Similarly, other benchmark systems use traces from the past that are reused to simulate the expected behaviour. An example of such a system is Linkbench [2], which provides a realistic and challenging test for persistent storage of social and web service data. The solution ensures testing of the reused spikes, but completely new and unexpected sudden spikes are out of the scope. In comparison, our work focuses on the imitations of different traces on the hand of the new parameters. In this way, new kinds of spikes that never occurred before and are difficult to anticipate in advance can also be simulated. Lu et al. [17] presented AutoFlow, a hotspot-aware system that supports dynamic load balance in distributed stream processing. AutoFlow contains a centralized scheduler, which monitors the load in the dataflow dynamically and implements state migrations accordingly. HotRing [5] is a hotspot-aware system that leverages a hash index, which provides fast access to hot objects by moving head pointers closer to them. These systems generate a flat workload and do not take into account the locality of objects.

In summary, benchmark frameworks described in this section are mostly designed to address different aspects (performance, scalability, availability, etc) of big data management systems as well as big data processing frameworks. As such, these systems mainly focus on generating flat workloads. Other recent works focus on dealing with spikes and variations in workloads. However, they fail to predict (unexpected) sudden spikes that have not occurred previously. In essence, none of these existing systems are designed to approximate hotspot behaviour in databases, particularly distributed NoSQL databases. This highlights and confirms the need for a configurable benchmark to mimic the hotspot behaviour in distributed databases, such as the workload proposed in this paper.

6 Conclusion

The current state of NoSQL benchmark systems does not appear to be sufficient to mimic spikes. To bridge this gap, we extended YCSB [6] by introducing a new workload that supports the generation of spikes with hotspot behaviour and uses a similar approach described by Bodik et al. [4]. Besides, a number of new parameters have been introduced in the workload, which makes it easy to generate the requested volume and/or data spikes. The workload has been validated, in comparison to the core workloads of YCSB on the default configuration of Cassandra and MongoDB databases. The results show that our proposed workload can be used to test the resilience of NoSQL databases caused by hotspot objects behaviour, which is currently lacking in the existing workloads of the YCSB benchmark system.

Acknowledgements. This research is partially funded by the Research Fund KU Leuven and the Cybersecurity Initiative Flanders (CIF) project.

References

1. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear Road: A Stream Data Management Benchmark (10 2004). https://doi.org/10.1016/B978-012088469-8/50044-9
2. Armstrong, T., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the Facebook social graph pp. 1185–1196 (2013)
3. Barahmand, S., Ghandeharizadeh, S.: Bg: A benchmark to evaluate interactive social networking actions. Citeseer (2013)
4. Bodik, P., Fox, A., Franklin, M., Jordan, M., Patterson, D.: Characterizing, modeling, and generating workload spikes for stateful services pp. 241–252 (2010). https://doi.org/10.1145/1807128.1807166
5. Chen, J., Chen, L., Wang, S., Zhu, G., Sun, Y., Liu, H., Li, F.: HotRing: A Hotspot-Aware In-Memory Key-Value Store. In: 18th USENIX Conference on File and Storage Technologies (FAST 20). pp. 239–252. USENIX Association, Santa Clara, CA (Feb 2020), https://www.usenix.org/conference/fast20/presentation/chen-jiqiang
6. 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 (2010)
7. Dayarathna, M., Suzumura, T.: XGDBench: A benchmarking platform for graph stores in exascale clouds pp. 363–370 (2012)
8. Dayarathna, M., Suzumura, T.: Benchmarking Graph Data Management and Processing Systems: A Survey. arXiv preprint arXiv:2005.12873 (2020)
9. Dey, A., Fekete, A., Nambiar, R., Rohm, U.: YCSB+T: Benchmarking web-scale transactional databases pp. 223–230 (2014)
10. Difallah, D.E., Pavlo, A., Curino, C., Cudre-Mauroux, P.: Oltp-bench: An extensible testbed for benchmarking relational databases. Proceedings of the VLDB Endowment **7**(4), 277–288 (2013)
11. Gao, W., Zhan, J., Wang, L., Luo, C., Zheng, D., Wen, X., Ren, R., Zheng, C., He, X., Ye, H., et al.: Bigdatabench: A scalable and unified big data and ai benchmark suite. arXiv preprint arXiv:1802.08254 (2018)

12. Ghazal, A., Ivanov, T., Kostamaa, P., Crolotte, A., Voong, R., Al-Kateb, M., Ghazal, W., Zicari, R.V.: Bigbench V2: the new and improved bigbench. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE). pp. 1225–1236. IEEE (2017)
13. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.A.: Bigbench: Towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD international conference on Management of data. pp. 1197–1208 (2013)
14. Gupta, P., Carey, M.J., Mehrotra, S., Yus, o.: Smartbench: A benchmark for data management in smart spaces. Proceedings of the VLDB Endowment **13**(12), 1807–1820 (2020)
15. Kumar, S.P., Lefebvre, S., Chiky, R., Soudan, E.G.: Evaluating consistency on the fly using YCSB. In: 2014 International Workshop on Computational Intelligence for Multimedia Understanding (IWCIM). pp. 1–6 (Nov 2014). https://doi.org/10.1109/IWCIM.2014.7008801
16. Leutenegger, S.T., Dias, D.: A modeling study of the TPC-C benchmark. ACM Sigmod Record **22**(2), 22–31 (1993)
17. Lu, P., Yuan, L., Zhang, Y., Cao, H., Li, K.: AutoFlow: Hotspot-Aware, Dynamic Load Balancing for Distributed Stream Processing. arXiv preprint arXiv:2103.08888 (2021)
18. Nambiar, Raghunath Othayoth and Poess, Meikel: The Making of TPC-DS. In: VLDB. vol. 6, pp. 1049–1058 (2006)
19. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: YCSB++: benchmarking and performance debugging advanced features in scalable table stores pp. 1–14 (2011)
20. PilHo, K.: Transaction processing performance council (TPC). Guide d'installation (2014)
21. Pirzadeh, P., Carey, M.J., Westmann, T.: BigFUN: A performance study of big data management system functionality. In: 2015 IEEE International Conference on Big Data (Big Data). pp. 507–514. IEEE (2015)
22. Pitman, J., Yor, M.: The two-parameter poisson-dirichlet distribution derived from a stable subordinator. The Annals of Probability pp. 855–900 (1997)
23. Poess, M., Smith, B., Kollar, L., Larson, P.: TPC-DS, taking decision support benchmarking to the next level
24. Sidhanta, S., Mukhopadhyay, S., Golab, W.: Dyn-YCSB: Benchmarking Adaptive Frameworks. In: 2019 IEEE World Congress on Services (SERVICES). vol. 2642-939X, pp. 392–393 (July 2019). https://doi.org/10.1109/SERVICES.2019.00119
25. TPC: Transaction Processing Performance Council ([Online] Available: tpcorg [Consulted on February 14, 2020]), http://www.tpc.org/
26. TPC-E: TPC-E is an On-Line Transaction Processing Benchmark. http://www.tpc.org/tpce/ (2020), online; accessed 20 Feburary 2021
27. Waudby, J., Steer, B.A., Karimov, K., Marton, J., Boncz, P., Szárnyas, G.: Towards Testing ACID Compliance in the LDBC Social Network Benchmark
28. Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., Madhyastha, H.V.: Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 292–308 (2013)
29. Xia, F., Li, Y., Yu, C., Ma, H., Qian, W.: BSMA: A Benchmark for Analytical Queries over Social Media Data. Proceedings of the VLDB Endowment **7**(13) (2014)