King Louie: Reproducible Availability Benchmarking of Cloud-hosted DBMS

Daniel Seybold
Institute of Information Resource
Management, Ulm University
Ulm
daniel.seybold@uni-ulm.de

Stefan Wesner
Institute of Information Resource
Management, Ulm University
Ulm
stefan.wesner@uni-ulm.de

Jörg Domaschka
Institute of Information Resource
Management, Ulm University
Ulm
joerg.domaschka@uni-ulm.de

ABSTRACT

Cloud resources have become a preferred operational model distributed Database Management Systems (DBMS) by offering the elasticity and virtually unlimited scalability, but increase the risk of failures with increasing cluster sizes. While distributed DBMS provide high-availability mechanisms, it is currently an open research question to what extent they are able to provide availability and performance guarantees in case of cloud resource failures. Especially as existing DBMS benchmarks do not consider availability. We present a comprehensive methodology for evaluating the availability of distributed DBMS in case of cloud resource failures. Based on this methodology, we introduce a novel framework that automates the full evaluation process, including the failure injection, and emphasizes reproducibility. The framework is validated by 16 diverse availability evaluations. The results show that distributed DBMS are not necessarily available even if sufficient replicas are available and clients can experience significant downtimes.

CCS CONCEPTS

• **Information systems** → **Database performance evaluation; Database utilities and tools**; • **Computer systems organization** → **Cloud computing**;

KEYWORDS

availability, NoSQL, cloud, distributed DBMS, benchmark

ACM Reference Format:

Daniel Seybold, Stefan Wesner, and Jörg Domaschka. 2020. King Louie: Reproducible Availability Benchmarking of Cloud-hosted DBMS. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20)*, March 30–April 3, 2020, Brno, Czech Republic. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3341105.3373968>

1 INTRODUCTION

The raise of cloud computing a decade ago has led to a change in the way storage backends are treated. Instead of large, single-instance servers, the NoSQL movement has established a large amount of different Database Management Systems (DBMS) that

share the concept of a distributed architecture [11]. Here, multiple independent DBMS instances form a cluster and provide the user with the impression of a single logical DBMS.

A distributed architecture enables scaling the DBMS horizontally as needed by adding more resources [26]. This increases the storage capacity of a DBMS as well as its processing capabilities. Often scaling can even be realised as an elastic scale-out during run-time without users of the DBMS experiencing any downtime [20].

Yet, the risk of failures increases with the size of the systems as failures are more likely for distributed DBMS than for single-instance DBMS. On the other hand, the distributed nature enables the introduction of replication by storing individual data items more than once. This facilitates a continued operation of the logical DBMS without the user noticing. However, the use of replication introduces overhead, as the various replicas of a data item need to be kept in sync; and due to the CAP theorem [9], availability of the DBMS and consistency of replicas are not independent, but influence each other. The higher the need for availability, the less consistency guarantees can be given in the case of failures. Vice-versa, the higher the need for consistency, the less available the system will be in the case of failures. Many of existing DBMS claim high-availability. Yet, it is widely unclear where in the continuum of possible consistency-availability trade-offs they reside and what will be the impact on the performance.

An established approach to exploit the scalability and elasticity properties of distributed DBMS is to host them across IaaS cloud systems, as they provide the necessary flexibility when more (virtual) hardware is required [26]. On the downside, cloud resources are volatile and also prone to failure on various levels [1, 16].

When designing a large-scale application, the choice of a storage backend is a crucial decision. Yet, no DBMS allows the direct configuration of availability properties; instead, a user often can only select indirect parameters such as replication degree and consistency demands, whose impact on availability is only mediately visible. Hence, it is difficult to assess the impact of a configuration on availability, but also to assess the impact of a failure on the overall DBMS performance. Moreover, future DBMS will experience different demands than existing ones as future IoT environments will expose constant write-heavy workloads that are contrary to the current read-heavy workloads.

In consequence, the choice of a DBMS requires extensive evaluations which are very time consuming when done manually, due to the large number of configuration properties and different failures [7]. Furthermore, these benchmarks need to be repeated for new DBMS appearing on the market, but also when new versions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAC '20, March 30–April 3, 2020, Brno, Czech Republic

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-6866-7/20/03...\$15.00

<https://doi.org/10.1145/3341105.3373968>

of existing DBMS are released. Hence, a large degree of automation is required for reproducible benchmarking [23].

To mitigate this situation, this paper provides the following contributions: (i) we define a reproducible and portable methodology for evaluating the availability of DBMS in the context of resource failures; (ii) we introduce a novel evaluation framework that implements the defined methodology; (iii) we present an Web- and IoT-based case study that validates the framework and provides novel insights into the availability of distributed DBMS.

Our considerations and also methodology focuses exclusively on the client perspective and hence, captures metrics such as performance (latency and throughput) and success rate of queries.

The remainder of this document is structured as follows: Section 2 presents background on distributed DBMS, availability, and cloud hosting. Section 3 discusses DBMS system models and failure models. Section 4 introduces the novel availability evaluation methodology. Section 5 presents the King Louie framework. Section 6 defines the case study while Section 7 discusses the respective results. Section 8 validates King Louie. Section 2 presents related work before Section 10 concludes.

2 BACKGROUND

This section summarises background on availability and distributed DBMS focusing on their high-availability mechanisms.

2.1 Availability

For computing systems, *availability* is defined as *the degree to which a system is operational and accessible when required for use* [14]. Besides *reliability* (the *measure of the continuity of correct service* [2]), availability is the main pillar of many fault-tolerant implementations [12]. In this paper, we focus on the availability aspect from the client perspective and assume that DBMS are reliable, *i.e.* work as specified. The availability of the DBMS can be affected by two kinds of conditions: (i) A high number of requests issued concurrently by clients, overloading the DBMS so that the requests of clients cannot be handled at all or are handled with an unacceptable *latency* $> \Delta t$. (ii) Resource failures occur that impact network connectivity or the compute/storage resources the DBMS is hosted on [12]. In this work, we solely consider (ii).

2.2 Distributed DBMS

Over the last decade the DBMS landscape is showing an increasing heterogeneity with respect to data models *relational*, *NoSQL* and *NewSQL*, but also with respect to distributed architectures [11, 22]. Distributed DBMS provide a logical DBMS instance to clients, but distribute the DBMS functionality across multiple physical or virtual resource entities. These resources hosting the DBMS nodes are connected via network and form a DBMS cluster.

Distributed DBMS provide horizontal scalability by exploiting sharding as a basic technique: the full data set is separated such that each data-hosting DBMS node manages only a local share of the overall data set. With more nodes joining the cluster, the overall capacity increases and the system scales horizontally.

As growing cluster sizes increase the probability of failures, distributed DBMS provide diverse means for replication [12]. A key characteristic is the *replication scope* that is classified into full

and partial replication [12]. Full replication defines that each DBMS node stores a full copy of the entire database. In partial replication, each data item may only be stored on a subset of all DBMS nodes, *i.e.* the replication factor is smaller than the DBMS cluster size. Partial replication closely relates to sharding as if sharding is used without replication, there is no tolerance against DBMS node failures. On the other hand, using replication without sharding means that all data is available on all DBMS nodes.

Due to given constraints of the CAP theorem [9] that a distributed storage system can only provide two out of the three properties: consistency (C), availability (A) and partition tolerance (P), distributed DBMS provide heterogeneous consistency guarantees [33]. Their evolutions show that these constraints are not a binary decision [8].

3 AVAILABILITY TRADE-OFFS

The heterogeneity of distributed DBMS offerings, their extensive amount of runtime configurations and the number of cloud resource offerings create a vast design space for operating distributed DBMS. While this design space includes use case specific decision such as the functional feature-driven DBMS selection [11] and the comparative evaluation of non-functional features such as performance and scalability [17, 21]. In this process the non-functional feature analysis needs to consider availability mechanisms as well to ensure a seamless operation in case of cloud resource failures [1, 16]. This work considers distributed DBMS from a cloud architect perspective with the primary focus on the client perspective and secondary on the DBMS operator perspective. As the clients main interest relies in getting the promised DBMS performance even in case of failures, we consider the performance impact in case of failures as key aspect in the following. Therefore, we do not discuss related approaches such as analysing the availability on the DBMS level [24] or evaluating the consistency impact on client and DBMS level [5].

In the following, we discuss availability and performance related design decision for selecting and operating cloud-hosted distributed DBMS, discuss potential cloud resource failure scenarios and classify both domains according to their impact on DBMS availability.

3.1 Operating Distributed DBMS

For operating distributed DBMS, we separate the design decisions into the *DBMS level* and *client level* and discuss the resulting *availability* and *performance* impact on the client level.

On DBMS level, a variety of fault-tolerance mechanisms are adopted in distributed DBMS [12]. Yet, the exposed runtime configurations of these fault-tolerance mechanisms are always DBMS specific and require the individual consideration and evaluation. A high-level set of common runtime configurations are the *cluster size*, the *replication factor*. With an increasing *cluster size* (CS), data is sharded across the DBMS nodes, an increase of the overall DBMS performance can be expected if the load is evenly distributed across all nodes in the cluster. In parallel, the probability of failures increases as DBMS nodes might be distributed across multiple potentially failing resources. Therefore, replication is preferably enabled to increase availability by defining a suitable *replication factor* (RF). The scale of the RF is limited by the CS or a DBMS specific *replication factor limit* (RFL) following the constraint $1 \leq RF \leq \min(CS, RFL)$.

Figure 1: Failure Recovery Steps

The replication factor defines the existing number of identical data items within a DBMS cluster. Thus, it increases data availability as in case of a DBMS node failure the remaining replicas still contain copies of the data. With respect to data accessibility, replicas can take over read and write requests depending on the applied client consistency as described below. Yet, increasing the RF comes with a cost in performance as the data replication and synchronization mechanisms add additional load on the DBMS cluster.

3.2 Cloud-hosted DBMS

As cloud resources tend to fail [1, 16], their usage impacts DBMS availability as failures on different levels of the cloud resource stack can have impact on individual DBMS nodes (running on a VM), but also on larger sets of them. For instance, the failure of a hypervisor will lead to the failure of all VMs on that hypervisor [28]. Consequently, the *failure type* can be caused on the physical level as servers, network links, network device, power supplies, or cooling may fail. As a result, all DBMS nodes hosted on these resources, either directly or virtualised, become unavailable. Similarly, on the software level, management software and hypervisors can fail; algorithms, network, and devices can be buggy, misconfigured, or in the process of being restarted. Any of these failures can affect virtual machines and virtual networks, but also physical servers, physical networks, entire racks, complete availability zones, or even entire data centres. The failure of a full *availability zone* or even an entire *region* leads to the unavailability of multiple physical servers and hence unavailability of all hosted virtual machines and consequently the DBMS nodes running on these resources. These failures can be classified based on their severity *permanent* and *temporary* and occurrence expressed in the *failure rate (FR)*.

In parallel, the elasticity of cloud resources enables the implementation of manual or (semi-) automated recovery mechanism to overcome such failures. In order to recover from such failures, we define a set generic recovery steps for cloud-hosted DBMS as depicted in Figure 1, one for temporary failures and one for permanent resource failures. While the cloud provider and DBMS specific steps can be executed manually or automated by relying on cloud orchestration tools (COTs) [4], the DBMS internal steps are executed by the DBMS itself. For temporary failures, depending on the failure type, only a subset of the depicted steps might be required, e.g. in case of a temporary network outage, the unavailable DBMS nodes will synchronize their data with the remaining nodes as soon as their are online again.

Figure 2: Availability Impact Factors

3.3 Impact Factor Classification

Operating distributed DBMS in the cloud and ensuring high-availability already becomes a challenging task due to the vast amount of distributed DBMS offers in combination with their heterogeneous runtime configuration [22]. The diverse set of potential cloud resource failures even increases this challenge. In Figure 2 we summarise availability and performance impact factors based on the *DBMS domain* and *cloud domain*. We classify the impact factors into *configurable* and *fixed* impact factors. Fixed impact factors are given by the selected DBMS such as architecture type or consistency model, or resource failure probabilities. Configurable impact factors offer a certain range of runtime configurations such as the cluster size for the DBMS or the provider for the cloud domain.

Consequently, the selection and operation process of cloud-hosted DBMS requires the consideration of a multitude of impact factors to ensure high performance, but also availability for the client level. Therefore, the preferred approach for underpinning this process is the comparative evaluation of suitable DBMS with respect to their non-functional features. But as the heterogeneous DBMS landscape raises new challenges in comparative DBMS evaluations [7], the volatile cloud context even increases these challenges [6] and raises the need for reproducible and portable evaluations [23]. Moreover, as current benchmarks only focus on performance related evaluation objectives, availability in the context of failures is not yet in the scope of DBMS benchmarks [27]. In order to address these limitations, we define a comprehensive methodology which enables comparative availability evaluation of cloud-hosted DBMS by explicitly considering the presented impact factors.

4 EVALUATION METHODOLOGY

In order to introduce the concepts and challenges of our availability evaluation methodology, we build upon the following baseline hypothesis that describes the expected client level performance and availability in case of a failure on DBMS level.

Hypothesis (H1): *If a DBMS node fails due to a cloud resource failure, the performance is temporarily decreasing as client requests to the failed node are stalled until the DBMS recognize the DBMS node failure and trigger the replicas for taking over these requests; with the replicas taking over the performance is increasing, but does not achieve the initial performance due to the reduced CS. In case the permanent recovery steps are executed, performance will decrease during the data synchronization/redistribution and increase afterwards to the initial*

Figure 3: Expected DBMS performance in case of failure

performance. Throughout these phases, the DBMS connection remains available and all DBMS request are executed successfully.

Based on H1, the performance undergoes four phases depicted in the corresponding Figure 3: the *healthy phase* represents the operational cluster with its initial CS; in the *unhealthy phase* n DBMS nodes are unavailable due to resource failures; in the *recovery phase* n recovery steps are executed to resolve the failures before the cluster reaches the *recovered phase* with finished data synchronization/redistribution and its initial CS of operational nodes.

4.1 Metrics

As comparative availability evaluations require a set of quantifiable metrics, we build upon the established DBMS performance metrics *throughput* and *latency* [15]. These metrics are correlated with the respective phases in order to enable significant availability evaluations. In the following we only build upon the throughput metric, but the resulting metrics can similarly applied for latency. In order to correlate the throughput with the specific phase in case of a failure, we define the *phase throughput (PT)* as follows:

Phase Throughput (PT): $\frac{\text{successful requests}}{\text{phase duration}}$

As availability metrics, we consider downtime and the *request success rate (RSR)* as relevant metrics. The downtime metrics build upon [28] and represent the minimal tolerable throughput in correlation to the respective phase. The *phase downtime (PD)* and the according *phase downtime duration (PDD)* are defined as follows:

Phase Downtime (PD): $PT < \text{threshold}$

PD Duration (PDD): $\sum_{\text{phase start}}^{\text{phase end}} PT < \text{threshold}$

The RSR builds upon the *expected service availability* [34] and spans over all all four phases:

Request Success Rate (RSR): $\frac{\text{successful requests}}{\text{total scheduled requests}}$

4.2 Evaluation Process

To define a comprehensive availability evaluation methodology that provides the introduced metrics, we define the evaluation process depicted in Figure 4. It comprises eight *evaluation tasks (ETs)* with the concept of explicit cloud failure injection (ET_5.1) in combination with recovery tasks (ET_5.2) to emulate realistic cloud failure scenarios [16]. ET_1 specifies of a comprehensive *evaluation scenario*, comprising the cloud resources, the specific DBMS with the availability-related configurations, a DBMS workload and the cloud failure to be injected with an optional recovery action. The actual execution of the evaluation starts with ET_2 by allocating the cloud resources, followed by the deployment of the DBMS (ET_3). The

Figure 4: Availability Evaluation Process

workload execution (ET_4) and the failure injection in combination with the recovery are executed in parallel and can be executed multiple times if specified in ET_1. After ET_4 is finished, the cloud resources need to be released (ET_6). The metric processing (ET_7) completes the execution phase of the evaluation process. In the final step ET_8, the results are analysed and additional or refined evaluation scenarios are defined for further executions.

4.3 Evaluation Process Execution

The execution of this evaluation process involves detailed knowledge of the cloud provider, DBMS and workload domain. In addition, the parallel ET_4 and ET_5 require an orchestrated execution. These constraints lead to a time consuming, error prone and technically challenging execution if done manually. This limits reproducibility and portability across the domains, e.g. executing the evaluation scenario for different cloud providers or DBMS. Consequently, a supportive framework is required to execute the evaluation process. By building upon established benchmarking guidelines [6, 7, 23] we define the following requirements towards a supportive availability evaluation framework to ensure significance and reproducibility:

Evaluation Design (R1): Reusable and comprehensive evaluation scenario specification need to be provided.

Abstraction (R2): A suitable abstraction level is required for the technical domain specific properties.

Failure Injection (R3): has to be supported for temporary and permanent failures of different failure types

Automation (R4): is required for all ETs of the execution phase to ensure reproducibility and transparency, including the orchestration of parallel but timely dependent ETs

Extensibility (R5): As each evaluation domain is constantly evolving, extensible specifications and architectures are required to support the integration of future domain specific constraints and failure models

Transparent Data (R6): the resulting data needs to comprise raw metrics and extensive meta-data such as monitoring data, task related details, DBMS and cloud resource details.

5 KING LOUIE FRAMEWORK

In order to implement the defined methodology, we present the novel evaluation framework King Louie that explicitly addresses the identified requirements R1 - R6. King Louie builds upon the Mowgli framework [29] that addresses the automated performance and scalability evaluation of cloud-hosted DBMS. King Louie supports ET_1

by providing JSON-based evaluation scenario templates¹ to describe comprehensive availability evaluation scenarios. A simplified example is detailed in Listing 1. The `dbms` object exposes all relevant details (cf. Figure 2) to deploy a DBMS cluster on arbitrary cloud resources, a comprehensive description of the deployment model is presented in [29]. The workload object specifies the desired workload, including the DBMS specific read/write consistency configurations. The `failureSpec` defines the desired failure to be injected and optional recovery actions to be executed. In its current state, King Louie supports the failure types `VM`, `AVAILABILITY_ZONE` and `REGION` and the recovery `ADD-NODE` for permanent failures and `RESTART-DBMS` for temporary failures.

King Louie builds upon a loosely coupled and extensible architecture (cf. Figure 5). Evaluation scenarios are submitted via the *Evaluation API* and the *evaluation orchestrator* maps the specified configurations to its technical implementations. These are provided by extensible cloud, DBMS and workload *domain catalogues* (R5). Based on these technical implementations, the *evaluation orchestrator* fully automates the evaluation process (R4), i.e. ET_2 – ET_7. In order to abstract cloud provider APIs and ease the DBMS deployment (R2), King Louie exploits cloud orchestration tools (COTs) [4], in particular the Cloudiator framework [3]. The workload is executed by separate *workload-API* (W-API) instances that are controlled by the *evaluation orchestrator*. The failure injection is controlled by the evaluation orchestrator by exploiting the COTs control over allocated cloud resources. This enables the injection of different failure types such as the (temporary) stopping or deletion of single VMs, but also for VMs in specific availability zones or even regions. Moreover, additional failure types can be injected on running VMs the evaluation orchestrator is able to execute custom commands on the deployed DBMS nodes via the COT (R3).

By orchestrating each ET, King Louie is able to process availability and performance metrics and add extensive meta-information such as runtime configurations, logs and monitoring data that are provided in machine interpretable formats for advanced post processing (R6). Currently, King Louie supports EC2 and OpenStack based clouds, seven DBMS and two different workloads².

Listing 1: Availability Scenario Template

```
{
  "dbms": {
    "type": "CASSANDRA",
    "instances": 3,
    "replicationFactor": 3,
    "vmLocation": "openstack",
    "vmType": "small",
    ...
  },
  "workload": {
    "type": "YCSB",
    "config": {
      "writeConsistency": "ONE",
      ...
    }
  },
  "failureSpec": [{
```

¹<https://omi-gitlab.e-technik.uni-ulm.de/mowgli/getting-started/tree/master/examples/availability>

²<https://omi-gitlab.e-technik.uni-ulm.de/mowgli/getting-started>

Figure 5: King Louie technical architecture

```
"failureType": "VM",
"severity": "permanent"
"failureRecovery": true,
"recoveryType": "ADD-NODE"
}]
}
```

6 CASE STUDY

We apply a case study to validate King Louie by analysing H1, focusing on the impact of cloud resource failures. Therefore, eight availability evaluation scenarios are specified, comprising Apache Cassandra (CA) and Couchbase (CB) in two cluster sizes (CS), a write-heavy IoT driven workload, a read-heavy Web driven workload and one failure specification. It is noteworthy that these specifications represent only a small frame of the supported evaluation scenarios supported by King Louie.

6.1 DBMS Specification

CA and CB are selected as both are popular NoSQL DBMS³ that promise high-availability and have already achieved promising results [19, 21]. Both provide a comparable multi-master architecture with a configurable replication factor. As they differ with respect to their consistency mechanisms, client-side consistency settings can not exactly be mapped for both DBMS. CA applies write ahead logging (WAL), while CB caches records directly in memory and persists them to disk asynchronously. The case study focuses on availability and performance optimized DBMS configurations by applying a replication factor of three and weak client consistency settings as detailed in Table 1. For CA, a write and read consistency of *ONE*⁴ is applied, while for CB the write consistency `replicateTo=0` (`R=0`) and `persistTo=0` (`p=0`) is applied, specifying the confirmation of a write as successful after the record is transmitted⁵. Conceptually, this setup overcomes two DBMS node failures. For CB, the automatic failover mechanism is explicitly enabled with the minimum configurable failover time of 30 seconds while in CA it is enabled by default with a default failover time of 10 seconds.

³<https://db-engines.com/en/ranking>

⁴https://docs.datastax.com/en/archived/cql/3.3/cql/cql_reference/cqlshConsistency.html

⁵<https://docs.couchbase.com/java-sdk/2.2/durability.html#persistto-and-replicateto>

Table 1: DBMS Runtime Specifications

Specification	Apache Cassandra (CA)	Couchbase (CB)
Version	3.11.2	5.0.1 community
Cluster size	3,7	3,7
Replication	3	3
Write consistency	ONE	P=0, R=0
Read consistency	ONE	default

6.2 Cloud Resource Specification

The cloud resource configurations for the DBMS nodes and W-API instance is detailed in Table 2. All physical hosts hosting VM_DBMS types use a two SSDs Raid-0 configuration. Each DBMS instance is configured to use 50% of the available memory for its operation.

Table 2: Cloud Resource Specifications

Specification	VM_DBMS	VM_WORKLOAD
Cloud	private OpenStack, version Rocky	
VM vCores	2	8
VM RAM	4GB	8GB
VM Disk	70GB	20GB
OS	Ubuntu Server 16.04	
network	private, 10GbE	

6.3 Workload Specification

The W-API of King Louie provides the Yahoo Cloud Serving Benchmark (YCSB) [10] in version 0.15.0 King Louie. The YCSB is selected as it is a widely adopted workload to evaluate distributed DBMS. Moreover, it supports a variety of distributed DBMS by implementing the client connection in a baseline approach without adding specialised code to improve availability on client side. In order to demonstrate the flexibility of our approach, we define two typical DBMS workloads, one write-heavy workload and one read-heavy workload as described in Table 3.

Table 3: Workload Specifications

Specification	write-heavy	read-heavy
Metric granularity	10 s	
Initial records	0	2.000.000
Record size	5KB	
Operations	8.000.000	
Operation type	100% write	95% read, 5% update
Read distribution	N/A	zipfian
client threads	4,16	4,16

6.4 Failure Specification

While King Louie supports an extensive number of potential failure scenarios, we define a baseline failure specification to get first insights into the availability of distributed DBMS. As presented

in Table 4, the failure specification comprises a permanent cloud resource failure on VM level and the respective recovery steps for permanent failures as depicted in Figure 1. Hereby, specific delays are specified to ensure a stable workload before the respective failure injection and recovery actions are triggered.

Table 4: Failure Specification

Specification	VM_FAILURE
Failure delay	180s write-heavy, 500s read-heavy
Failure type	VM
Severity	permanent
Recovery delay	180s
Recovery type	add new DBMS node

7 AVAILABILITY EVALUATION RESULTS

The case study results in 16 availability scenarios, *i.e.* 2 DBMS types * 2 cluster sizes * 2 workload types * 2 workload intensities * 1 failure. Each availability scenario is executed 10 times, resulting in 160 data sets. Each data set contains raw performance metrics, phase-specific throughput metrics for additional processing, monitoring data and auto-generated plots. Due to space constraints, we omit the presentation of the 16 threads workload intensity, but to ensure transparency, all collected evaluation data is publicly available [31].

First, for each data set, an explorative analysis based on the PT is executed by comparing the throughput graphs with H1. Based on this comparison, a set of reoccurring throughput development patterns are identified for each evaluation scenario. If different patterns are identified, the data sets are classified into the resulting pattern. For each identified pattern per availability scenario, one exemplary plot is provided in while all plots are available online⁶. In the following, we term each identified availability scenario pattern according the scheme *DBMS TYPE_CLUSTER SIZE_PATTERN ID*,

Second, for each availability scenario a descriptive analysis is carried out, resulting in measurement tables. Each Table contains the introduced phase-specific metrics (*cf.* Section 4.1) by providing the average phase throughput (APT) including the standard deviation. The downtime threshold for all evaluation scenarios is set to 10 ops/s, *i.e.* each reported throughput value below 10 ops/s is marked as downtime and the resulting downtime duration (PDD) is provided. In addition, the occurrence ratio (OR) of each pattern over the ten executions per availability scenario is provided. The RSR is provided as average over each 10 executions.

7.1 Apache Cassandra Results

7.1.1 Write-heavy Workload. The explorative analysis of the CA for the write-heavy workload results in two patterns as shown in Figure 6. Hereby, P1 and P2 reoccur for both cluster sizes. The graphs in Figure 6 clearly show that none of the identified patterns matches the expected hypothesis as for both cluster sizes a downtime occurs after the VM failure is injected. Moreover, the downtime varies from 10s to 70s as described in the corresponding measurement Table 5. Hereby, the OR of the 10s downtime, *i.e.* CA-CS3-P1 and CA-CS7-P1, is significantly higher than 70s downtime

Figure 6: Cassandra patterns for write-heavy workload

patterns, *i.e.* CA-CS3-P2 and CA-CS7-P2. The fact that the throughput in the unhealthy phase remains on a similar or even slightly higher average compared to the healthy phase relates to the applied weak write consistency and replication factor of three. As in the unhealthy phase, one replica is missing, less internal replication operations are executed while there are still enough replicas available to satisfy the applied write consistency. The decreased throughput during the recovery phases relates to the redistribution of data to the newly added DBMS node. Moreover, this finding is validated by existing elasticity studies [21, 30]. The throughput variations during the recovered phase (*cf.* Table 5) are caused by internal compaction mechanisms and potential resource interferences due to the fact that the I/O bandwidth of the physical server hosting the DBMS VMs is shared with multiple tenants.

In order to analyse the unexpected throughput development, the raw YCSB results are checked for potential issues on client-side, but none are present in the result files. While P1 is partially explainable as 10s is the default request timeout before a new DBMS node takes over⁶, this should only affect the client threads trying to write to the unavailable DBMS node and consequently the throughput should only decrease for their stalled write requests while the results clearly show that the overall throughput reaches the downtime threshold. Moreover, the less frequent P2 can not be correlated to Apache Cassandra configuration options.

7.1.2 Read-heavy Workload. The read-heavy result also show two throughput patterns P1 and P2 for both cluster sizes as depicted in Figure 7. Again, these patterns do not match the expected throughput development of H1, they even show an increased downtime compared to the write-heavy results. As Figure 7 depicts and the according measurement Table 5 shows, the downtime ranges from 10s to 240s, spanning over the whole unhealthy, recovery and even into the recovered phase. Again the YCSB raw results do not show any issues. While the 10s downtime is again partially explainable due to the default request timeout⁷, the 240s downtime does not correlate with any CA configuration settings on client and DBMS side. A second deviation from the expected throughput development, the increased APT in the recovered phase compared to the initial phase is noteworthy. The explanation refers to the workload type as for the read-heavy workload, the longer phase duration of the recovered phase improves the caching of frequently accessed

⁶<https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/operations/opsRepairNodesHintedHandoff.html>

Figure 7: Cassandra patterns for read-heavy workload

records. Compared to the write-heavy workload, the impact of additional cloud tenants is barely present as the read-heavy workload is mainly memory- and compute-bound and not I/O-bound.

7.2 Couchbase Results

7.2.1 Write-Heavy Workload. For CB three patterns are identified for the write-heavy workload, CB-CS3-P1, CB-CS3-P2 and CB-CS7-P1 as depicted in Figure 8 and the corresponding measurement Table 6. The three node patterns, CB-CS3-P1 and CB-CS3-P1, show that the VM failure causes an unexpected availability of the DBMS on the client-side, either with the beginning of the recovery phase (CB-CS3-P1) or even before the recovery phase (CB-CS3-P2). The resulting YCSB results show frequent occurrence of timeout exceptions that cause the termination of the workload even as enough replicas are available. For seven node cluster, CB-CS7-P1, the availability of the cluster is provided with the reoccurring pattern shown in Figure 8c. Hereby, a downtime of 30s is identified that correlates to the configured take-over time of CB. CB-CS7-P1 also shows a reoccurring throughput decrease in the recovered phase indicates due to ongoing CB internal management processes. The results indicate that for a write-heavy workload, a CB size of three nodes is not sufficient to overcome one node failure, even with a weak write consistency applied.

7.2.2 Read-Heavy Workload. The identified patterns of the read-heavy workload are depicted in Figure 9 and the correlating Table 6. Compared to the write-heavy workload patterns, they do not show any persistent downtime. Pattern CB-CS3-P1 shows a downtime of 30s and the less frequent CB-CS3-P2 a downtime of 175s (*cf.* Table 6). For the seven node cluster, only the pattern CB-CS7-P1 with the downtime of 30s is identified. Hereby, the 30s downtime matches again the configured take-over time, but also shows that a DBMS node failure causes a downtime for all clients, even if replicas could handle the read requests. Regarding the 175s downtime, the YCSB result files show temporary failure exceptions during the recovering phase and disappear in the recovered phase.

7.3 Threats to Validity

Even with the support of King Louie, the complexity of the cloud, DBMS and workload domain requires the consideration of the following aspects for adopting the results. For cloud domain, the recovery phase duration can be influenced by provider dependent VM

Table 5: Cassandra Phase Measurements

Phase	Metric	write-heavy				read-heavy			
		CA-CS3-P1	CA-CS3-P2	CA-CS7-P1	CA-CS7-P2	CA-CS3-P1	CA-CS3-P2	CA-CS7-P1	CA-CS7-P2
all	OR	7/10	3/10	9/10	1/10	7/10	3/10	7/10	3/10
all	RSR	99.99%	99.99%	99.99%	99.99%	99.65%	99.74%	100.00%	99.99%
initial	APT (σ)	4305 (80)	4254 (71)	4718 (64)	4779 (0)	1399 (157)	1345 (68)	2950 (89)	2816 (99)
unhealthy	APT (σ)	4384 (266)	2886 (215)	4518 (60)	3248 (0)	1728 (133)	8 (1)	3336 (41)	27 (1)
	PDD	10s	70s	10s	70s	10s	240s	10	240
recovering	APT (σ)	3361 (251)	2746 (106)	4324 (71)	4448 (0)	1967 (111)	1 (0)	3512 (42)	2 (0)
recovered	APT (σ)	4301 (490)	4071 (48)	4364 (143)	4262 (0)	2179 (152)	1855 (57)	3650 (21)	2928 (42)

Figure 8: Couchbase patterns for write-heavy workload

Figure 9: Couchbase patterns for read-heavy workload
Table 6: Couchbase Phase Measurements

Phase	Metric	write-heavy			read-heavy		
		CB-CS3-P1	CB-CS3-P2	CB-CS7-P1	CB-CS3-P1	CB-CS3-P2	CB-CS7-P1
all	OR	7/10	3/10	10/10	8/10	2/10	10/10
all	RSR	23.61%	18.19%	99.99%	99.99%	99.99%	99.99%
initial	APT (σ)	5172 (136)	6130 (96)	4718 (64)	3205 (692)	3691 (600)	4975 (140)
unhealthy	APT (σ)	3219 (77)	17 (24)	4925 (86)	2470 (525)	100 (22)	4643 (216)
	PDD	30s	∞	30s	30s	175s	30s
recovering	APT (σ)	82 (65)	0	5554 (57)	2581 (801)	867 (250)	5360 (403)
recovered	APT (σ)	0	0	4592 (52)	3205 (697)	3489 (843)	6213 (170)

spawning times and the DBMS performance in general can be impacted by resource interferences on different levels. As King Louie supports multiple cloud providers, these challenges can easily be

addressed by additional case studies. With respect to the DBMS domain, vanilla DBMS installations are applied to ensure comparable results but there are numerous DBMS specific configurations that

can affect the availability results. Consequently, additional DBMS specific case studies need to be executed. As the DBMS catalogues of King Louie are easily extensible, DBMS specific configurations can be integrated for DBMS specific studies. For the workload domain, the usage of the YCSB enables comparability by supporting multiple DBMS, reporting DBMS performance metrics and failed requests. Yet, the implementation for the DBMS connections solely rely on the DBMS drivers and follow a basic approach without applying any dedicated availability mechanisms on client side. Consequently, the results represent a baseline and pave the path to evaluate enhanced implementations on client side as the YCSB is easily extensible and King Louie also supports custom workloads via the W-API.

Finally, the case study focuses on the client side while availability on the DBMS side is another important aspect. The 80 data sets show that even after the recovery phase, the cluster state can remain unhealthy. This is reported two times for CA and eight times for CB. These results are repeated and the unhealthy ones are excluded from the presented results. Hence, the root cause analysis of these cluster states requires further investigation on the DBMS side.

7.4 Lessons Learned

Based on the case study results, we provide a set of lessons learned on the availability of cloud-hosted DBMS in the case of failures: (i) as none of the results match the expected H1, there is even an increasing need for King Louie to further evaluate potential impact factors; (ii) the reproducibility of King Louie enables the identification of the different availability patterns in the first place; (iii) resource failures might not affect the availability on the DBMS side, but on the client side severe downtimes can occur or even client connections can be interrupted; (iv) potential throughput drops or even temporary downtimes need to be taken into account by client applications and respective mechanisms need to be in place to handle such downtimes; (v) as the results also show only particular explainable results, additional availability evaluations are required, considering DBMS specific configurations and metrics; (vi) even building upon established DBMS, guaranteeing DBMS availability and ensuring certain client performance is a challenging task that becomes even more challenging for automated operation as preferred in the cloud. Hence, comprehensive additional research with respect to DBMS behaviour and suitable COTs is required.

8 KING LOUIE VALIDATION

This section validates King Louie features against the introduced requirements and discusses potential limitations and extensions.

Evaluation design (R1) is enabled by providing comprehensive evaluation scenario templates that enable reusability and multiple customization parameters¹. In addition, a sequence of multiple failures including optional recovery mechanism can be specified. *Abstraction (R2)* is addressed by building upon COTs for abstracting the communication with cloud provider APIs and the usage of DBMS and workload catalogues to abstract the technical details for deploying the DBMS and executing workloads. King Louie supports the specification of custom properties in the evaluation scenario, but the respective technical implementation needs to be able to process these custom properties. Its abstraction is shown by applying the availability evaluation for two DBMS of different cluster sizes

and workload types. Its abstraction of different cloud providers has been validated in previous work [29]. *Failure injection (R3)* is validated by the case study by applying the VM failure type including the corresponding recovery mechanisms. While King Louie provides the framework for arbitrary failure injection, it currently supports the injection of failures on the VM and availability zone level. Additional failures can be easily added by implementing a single interface. *Automation (R4)* is demonstrated by executing 160 evaluation scenarios, generating the data sets for post-processing the availability metrics. Yet, the post-processing is not automatically executed by King Louie. *Extensibility (R5)* is addressed by building upon a loosely coupled architecture and applying the concept of domain-specific catalogues. This results in extensibility on different levels as adding or customizing DBMS deployments only requires the update of the respective scripts. Adding new failure types only requires the implementation of an interface in the evaluation orchestrator. *Transparent data (R6)* to ensure significance and reproducibility is provided by extensive data sets comprising raw performance metrics, but also monitoring data and execution logs to enable the full traceability of each evaluation execution [31].

As the availability evaluation of distributed DBMS in the volatile cloud context is a challenging task that requires the technical knowledge of multiple domains, King Louie provides a novel framework to support such evaluations by enabling reproducibility and portability. As the case study results show, there is a significant need in the thorough availability evaluation of cloud-hosted DBMS in order to provide high-available cloud services.

9 RELATED WORK

The evolving heterogeneity of distributed DBMS [11, 12, 22] has lead to numerous comparative evaluations of cloud-hosted distributed DBMS. A variety of DBMS workloads have been established for evaluating distributed DBMS in the context of Web and Big Data applications [25, 27]. Evaluations building upon these workloads focus either on the sole DBMS performance and scalability [17], the performance impact of different cloud resources [21]. With respect to availability evaluations, only the aspect of overload situations due to increasing workload intensities are evaluated [20, 30]. Yet, these studies do not consider the aspect of resource failures.

As the number of cloud resource offerings and distributed DBMS is still increasing, supportive evaluation frameworks are required to enable reproducible and portable evaluations of cloud-hosted DBMS [7, 23, 27]. Yet, none of them provides the capabilities of cloud resource failure injection and consequently none of them supports the evaluation of the introduced availability metrics.

A first approach towards analysing the availability of distributed DBMS is presented by [13] by studying existing DBMS bugs with respect to failure recovery mechanisms. Yet, the study focuses only on the DBMS layer and does not support failure injection nor the evaluation of client-side availability metrics. Building upon the "chaos monkey" concept, [32] present a framework to inject random cloud resource failures into cloud-hosted applications. While King Louie builds upon this concept, [32] does not evaluate any availability metrics nor is it intended for DBMS as failures are injected randomly which hinders reproducible and comparable evaluations. The Gremlin approach [18] presents a failure injection framework

for evaluating the availability of micro-services. A failure model describes the failures on the network level and injects them into a running system, analysing the latency impact on the client-side. Yet, Gremlin only considers stateless applications network failures, without focusing on stateful services and resource failures.

Consequently, a supportive framework for evaluating the availability of distributed DBMS under different cloud resource failures is missing. Therefore, King Louie addresses these limitations by enabling reproducible and portable availability evaluations for distributed DBMS by supporting cloud resource failure injection.

10 CONCLUSION

Cloud resources are a preferred solution to operate distributed DBMS. Yet, with increasing DBMS cluster sizes, the probability of cloud failures increases. Therefore distributed DBMS implement heterogeneous availability mechanisms and expose a diverse set of runtime configurations. Yet, in case of cloud resource failures, it remains unclear to what extent these mechanisms are able to ensure availability and guarantee stable performance. As supportive benchmarks are missing, we define a comprehensive availability evaluation methodology that ensures reproducibility and portability. Based on this methodology, we introduce the novel King Louie framework that automates the full evaluation process, including the cloud failure injection. We validate King Louie in a client-centric case study for different DBMS, cluster sizes and workload types resulting in 16 evaluation scenarios with 160 data sets. The results show novel and unexpected insights with respect to the availability and performance development in case of cloud resource failures.

Future work will focus on advanced evaluations that comprise multiple cloud failures and recovery steps. Moreover, machine learning techniques will be applied to derive efficient failure mitigation strategies. Also, King Louie will be applied to evaluate the availability from the DBMS side, including data related aspects such as accessibility and consistency.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the Federal Ministry of Education and Research of Germany under grant agreement 01IS18068 (SORRIR) and the EC's Framework Programme HORIZON 2020 under grant agreements 731664 (MELODIC) and 732667 (RECAP).

REFERENCES

- [1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58.
- [2] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing* 1, 1 (2004), 11–33.
- [3] D. Baur, D. Seybold, F. Griesinger, H. Masata, and J. Domaschka. 2018. A Provider-Agnostic Approach to Multi-cloud Orchestration Using a Constraint Language. In *CCGRID*. 173–182.
- [4] Daniel Baur, Daniel Seybold, Frank Griesinger, Athanasios Tsitsipis, Christopher B. Hauser, and Jörg Domaschka. 2015. Cloud Orchestration Features: Are Tools Fit for Purpose?. In *IEEE/ACM UCC*.
- [5] David Bermbach. 2014. *Benchmarking eventually consistent distributed storage systems*. KIT Scientific Publishing Karlsruhe.
- [6] David Bermbach, Erik Wittern, and Stefan Tai. 2017. *Cloud service benchmarking*. Springer.
- [7] Lexi Brent and Alan Fekete. 2019. A Versatile Framework for Painless Benchmarking of Database Management Systems. In *Australasian Database Conference*. Springer, 45–56.
- [8] Eric Brewer. 2012. CAP twelve years later: How the “rules” have changed. *Computer* (2012).
- [9] Eric A. Brewer. 2000. Towards robust distributed systems. In *PODC*.
- [10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *ACM SoCC*.
- [11] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 40.
- [12] Jörg Domaschka, Christopher B. Hauser, and Benjamin Erb. 2014. Reliability and availability properties of distributed database systems. In *Enterprise Distributed Object Computing Conference (EDOC), 2014 IEEE 18th International*. IEEE, 226–233.
- [13] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 539–550.
- [14] Anne Geraci, Freny Katki, Louise McMonegal, Bennett Meyer, John Lane, Paul Wilson, Jane Radatz, Mary Yee, Hugh Porteous, and Fredrick Springsteel. 1991. *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries*. IEEE Press.
- [15] Jim Gray. 1992. *Benchmark handbook: for database and transaction processing systems*.
- [16] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffrey Adityatama, and Kurnia J. Eliazar. 2016. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 1–16.
- [17] Abdeljawad Hendawi, Jayant Gupta, Liu Jiayi, Ankur Teredesai, Ramakrishnan Naveen, Shah Mohak, and Mohamed Ali. 2018. Distributed NoSQL Data Stores: Performance Analysis and a Case Study. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 1937–1944.
- [18] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems*. IEEE.
- [19] Murtadha Al Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Ly-chagin, Ian Maxon, and Till Westmann. 2019. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2275–2286.
- [20] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. 2011. On the elasticity of NoSQL databases over cloud management platforms. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM, 2385–2388.
- [21] Jörn Kuhlenskamp, Markus Klems, and Oliver Röss. 2014. Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1219–1230.
- [22] Somnath Mazumdar, Daniel Seybold, Kyriakos Kritikos, and Yiannis Verginadis. 2019. A survey on data storage and placement methodologies for Cloud-Big Data ecosystem. *Journal of Big Data* 6, 1 (2019), 15.
- [23] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, P. Tüma, and A. Iosup. 2019. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. *IEEE Transactions on Software Engineering* (2019), 1–1.
- [24] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 117 (Oct. 2019), 28 pages.
- [25] Vincent Reniers, Dimitri Van Landuyt, Ansar Rafique, and Wouter Joosen. 2017. On the state of nosql benchmarks. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM, 107–112.
- [26] Sherif Sakr. 2014. Cloud-hosted databases: technologies, challenges and opportunities. *Cluster Computing* 17, 2 (2014), 487–502.
- [27] Daniel Seybold and Jörg Domaschka. 2017. Is Distributed Database Evaluation Cloud-Ready?. In *ADBIS*. Springer, 100–108.
- [28] Daniel Seybold, Christopher B. Hauser, Simon Volpert, and Jörg Domaschka. 2017. Gibbon: An Availability Evaluation Framework for Distributed Databases. In *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*. Springer, 31–49.
- [29] Daniel Seybold, Moritz Keppler, Daniel Gründler, and Jörg Domaschka. 2019. Mowgli: Finding your way in the DBMS jungle. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*. ACM, 321–332.
- [30] Daniel Seybold, Nicolas Wagner, Benjamin Erb, and Jörg Domaschka. 2016. Is elasticity of scalable databases a Myth?. In *IEEE Big Data*.
- [31] Daniel Seybold, Stefan Wesner, and Jörg Domaschka. 2019. King Louie: DBMS Availability Evaluation Data Sets. <https://doi.org/10.5281/zenodo.3459604>
- [32] Ariel Tseitlin. 2013. The Antifragile Organization. *Commun. ACM* 56, 8 (2013).
- [33] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 5.
- [34] Ming Zhong, Kai Shen, and Joel Seiferas. 2008. Replication Degree Customization for High Availability. In *EuroSys*.