BenchFoundry: A Benchmarking Framework for Cloud Storage Services

David Bermbach[1(✉)], Jörn Kuhlenkamp[1], Akon Dey[2,4], Arunmoezhi Ramachandran[3], Alan Fekete[4], and Stefan Tai[1]

[1] Information Systems Engineering Research Group, Technische Universität Berlin, Berlin, Germany
{db,jk,st}@ise.tu-berlin.de
[2] Awake Security Inc., Mountain View, CA, USA
akon@awakesecurity.com
[3] Tableau Software Inc., Palo Alto, CA, USA
arunmoezhi@gmail.com
[4] University of Sydney, Sydney, Australia
{akon.dey,alan.fekete}@sydney.edu.au

Abstract. Understanding quality of services in general, and of cloud storage services in particular, is often crucial. Previous proposals to benchmark storage services are too restricted to cover the full variety of NoSQL stores, or else too simplistic to capture properties of use by realistic applications; they also typically measure only one facet of the complex tradeoffs between different qualities of service. In this paper, we present BenchFoundry which is not a benchmark itself but rather is a benchmarking framework that can execute arbitrary application-driven benchmark workloads in a distributed deployment while measuring multiple qualities at the same time. BenchFoundry can be used or extended for every kind of storage service. Specifically, BenchFoundry is the first system where workload specifications become mere configuration files instead of code. In our design, we have put special emphasis on ease-of-use and deterministic repeatability of benchmark runs which is achieved through a trace-based workload model.

Keywords: Cloud storage services · Benchmarking · Quality of service

1 Introduction

The ability to assess the quality of a service is of great importance in any service-oriented application architecture. Naturally, a variety of techniques have been proposed to this end. Many collect basic monitoring data for a specific quality like performance while some may also include user ratings. Others focus on a specific objective such as formalization in SLAs or service composition in business processes. Surprisingly, little attention has been paid to assessing services by running arbitrary application-driven workloads in a distributed deployment (which is in some cases required by measurement approaches, e.g., [4], but also

M. Maximilien et al. (Eds.): ICSOC 2017, LNCS 10601, pp. 314–330, 2017.
https://doi.org/10.1007/978-3-319-69035-3_22

a prerequisite for benchmark scalability) while measuring multiple qualities at the same time. For this, different application-driven workloads are necessary to impose different kinds of stress to the service under consideration. Distribution-aware quality assessments are needed to reveal otherwise undiscoverable insights. Additionally, each single quality should also be seen in the context of other, potentially conflicting qualities and their particular trade-offs.

In this paper, we will focus on cloud storage services. Today, the sheer number of available cloud storage services and database systems is staggering – in May 2017, nosql-databases.org lists more than 225 NoSQL database projects, a number that does not even include traditional relational database systems and services (RDBMS). Selecting a service from this extensive set for an application scenario requires an understanding of at least two main criteria: (a) functionality, i.e., implemented features, data model, etc., and (b) non-functional properties, i.e., the system qualities provided by the storage service. In this paper, we will focus on the comparability of cloud storage services in terms of quality. We suggest a novel benchmarking approach and middleware to provide the necessary insights into this: For our purposes, a benchmark is a standard workload that is applied to the system or service under test (SUT) while a standard set of measurements are collected in a standard way. For example, the Transaction Processing Council (TPC) has defined TPC-E representing the workload of a brokerage firm, to evaluate on-line transaction processing performance by metrics such as transactions-per-second in relational database systems. The term micro-benchmark is used when the workload does not have the entire range of features of a realistic application, but is limited to exploring the sensitivity to key variables in the workload characteristics [6].

There is a plethora of previous work, not only in general service quality assessment but especially on database benchmarking. However, existing database benchmarking approaches have severe disadvantages: Some approaches, e.g., TPC benchmarks or OLTPBench [9], have strict functional and non-functional requirements on supported database systems which are currently only fulfilled by RDBMS, e.g., Amazon RDS[1]. As such, these benchmarks cannot be used to study NoSQL systems such as Amazon's DynamoDB[2] or S3[3] services. Other approaches, e.g., YCSB [8] or YCSB++ [14] are essentially micro-benchmarks [6]. While these are useful for understanding how tiny changes in workloads affect system quality, they rarely mimic realistic application workloads. Existing approaches are also lacking in regards to extensibility of workloads, multi-quality measurements, (geo-)distribution support of the benchmark out of the box, fine-grained result collection, or ease-of-use. Finally, database benchmarking typically focuses on the database system rather than the service(s) that the database system provides – an important detail when it comes to assessing quality also from a service consumer perspective.

[1] aws.amazon.com/rds.
[2] aws.amazon.com/dynamodb.
[3] aws.amazon.com/S3.

Today, application developers often face a significant challenge: to implement the benchmark for all storage services of interest from scratch. Addressing this real-world concern, we present in this paper the result of designing and actually implementing ideas from our previous vision paper [3]: BenchFoundry is not a benchmark itself, rather it is a benchmarking framework which can execute arbitrary application-driven benchmark workloads in a distributed deployment, measure multiple qualities at the same time, and can be used or extended for every kind of storage service. Specifically, BenchFoundry is the first system in this domain where workload specifications become mere configuration files instead of code. In our design, we have put special emphasis on ease-of-use and deterministic repeatability of benchmark runs which is achieved through a trace-based workload model.

One contribution of this paper is to capture detailed requirements or desirable features of benchmarks for cloud services such as storage services; this is in Sect. 2 along with related work. Another contribution is the BenchFoundry proposal as a way to meet these requirements; Sect. 3 gives the high-level overview and Sect. 4 some implementation details. Our final contribution is to evaluate BenchFoundry (Sect. 5) by showing how some requirements are met during case-study experiments. We also discuss limitations and effects of design choices in our approach (Sect. 6).

2 Modern Storage Service Benchmarking

In this section, we use our extensive experience in cloud service benchmarking, e.g., [6], to identify requirements for modern benchmarks in general (and for benchmarking cloud storage services in particular) including their implementations. We also discuss existing work in this field.

> **(R1) Multi-Quality:** *Benchmarks should measure all sides of a particular tradeoff. Exceptions are only permissible where the respective other qualities are comparable; this should be verified by another benchmark.*

Traditionally, database benchmarking has mainly been done for performance evaluation, e.g., through TPC[4] benchmarks or with YCSB [8]. Over the last few years, some approaches have been developed for consistency benchmarking with varying degrees of meaningfulness[5], e.g., [2,4,14,17], security impacts on performance, e.g., [13], as well as an open source project for testing ACID isolation guarantees[6]. However, these are all more or less single quality benchmarks. Still, measuring more than one quality at the same time is crucial since modern distributed database systems and services are inherently affected by tradeoffs [1,4] – being top ranked for one quality is trivial when disregarding the

[4] tpc.org.

[5] One of the core requirements for benchmarks is to use meaningful and understandable metrics as well as to offer relevant results to a broad target audience [3,10–12].

[6] github.com/ept/hermitage.

respective other qualities. To make such tradeoff decisions transparent, modern benchmarking should always imply multi-quality benchmarking.

(R2) No Assumptions: *Benchmarks should make as little assumptions on the service under test (SUT) as possible. Instead an ideal case should be identified, deviations tolerated and measured as additional quality metrics for broad applicability and benchmark portability.*

Existing benchmark tools often have strict functional and non-functional requirements on supported storage services, e.g., requiring transactional features with strict ACID guarantees [9]. However, it would be preferable to reach a broader applicability and stronger portability [10,12] by transforming such strict requirements into measured qualities instead. For instance, transactions could also be executed in a best-effort way while tracking ACID violations as an additional quality metric.

(R3) Realistic Workloads: *Benchmarks should use realistic application-driven workload that mimick the target application as closely as possible.*

Micro-benchmarks certainly have their benefits for some use cases: they are a perfect fit for studying how a system reacts to small workload changes or to test isolated features. They are also easier to implement. However, the relevance of benchmarking results for a given application depends on the similarity of application workload and benchmarking workload – the greater the difference the less relevant are results. Therefore, application-driven benchmarking with realistic workloads that emulate the given use case as close as possible is typically preferable over synthetic micro-benchmarks like YCSB.

(R4) Extensibility: *Benchmarks should be extensible and configurable to account for future application scenarios and new storage services.*

Modern applications evolve at an as yet unheard of pace. As such, modern benchmarking tools need to be extensible and configurable: They must be able to support changes in benchmark workloads which reflect new application developments as well as new storage services which do not exist at the time of designing the benchmark. Typically, this is achieved through adapter mechanisms and suitable abstractions, e.g., in [8]. However, these abstractions should be carefully chosen, e.g., the data model of YCSB is obviously focused on column stores, which makes it a less than perfect fit for other kinds of storage services. We believe that a modern benchmark should distinguish a logical and physical data model in its adapter layer.

(R5) Distribution: *Benchmarks should always be distribution-aware and implementations should come with the necessary coordination logic for running multiple instances in parallel.*

Modern applications as well as underlying storage services are inherently distributed or even geo-distributed. Consequently, a modern benchmark should also be designed for distribution and its implementation should build on measurement clients that can be distributed. Parallelization through distribution is also important when measuring the scalability of storage services or simply for benchmarking a service that is already at scale (scalability of the benchmark tool). Also, some benchmarking approaches heavily rely on distributed execution, e.g., [4]. However, distributing workloads is a challenging problem, e.g., asserting that inserts precede updates to the same database key.

(R6) Fine-Grained Results: *Benchmarks should always log fine-grained results, they should never voluntarily delete information.*

Often, benchmarking tools only report aggregated results, e.g., [8]. While this is convenient for reporting purposes, this effectively loses a wealth of information: results such as the saw pattern or the night/day pattern from [4] would never have been found if the available information were only aggregates or even a CDF. Therefore, benchmarking tools should log detailed results at operation level, i.e., for each operation the outcome, start and end timestamp, retrieved results of read requests, etc.

(R7) Deterministic Execution: *Benchmarks should be able to deterministically re-execute the exact same workload.*

A key aspect of benchmarking is repeatability, i.e., repeating a benchmark run several times should yield identical or comparable results. In this regard, all benchmarking approaches known to the authors have a fundamental problem: they randomly select database keys and generate data at benchmark runtime. While such an approach has obvious benefits, it also means that repeated executions may not always yield comparable results or that seemingly comparable results may in fact have been produced by fundamentally different workloads. When using such implementations, the only way to counter this effect to a certain degree is to use long-running experiments, up to several hours or even days, or to carefully inspect the generated data afterwards (which, however, due to the unavailability of detailed results (R6) is typically not possible). We believe, therefore, that modern benchmarks should be trace-based, i.e., should be able to replay a given workload in a fully deterministic way. In a distributed workload generator, fixing the seed for randomization is not sufficient for producing a deterministic workload, due to the non-deterministic speeds of execution across multiple machines.

(R8) Ease-of-use: *Benchmarks should have ease-of-use as a core focus.*

A benchmark should focus on ease-of-use to foster adoption and use. Often, it is not possible to benchmark all services – relying on results of third parties may be an option. However, this is only possible in case of widespread use of the specific benchmark and also depends on the willingness of people to share their

results. Setting up open source systems is often a tedious exercise; we, therefore, believe that a core design focus of benchmark tools should be on ease-of-use. Obviously, this requires benchmarks to also come with an implementation as done for the more recent TPC benchmarks. Ease-of-use is also emphasised by Seybold and Domaschka [16].

3 BenchFoundry Design and Architecture

In BenchFoundry, we address each of the requirements from Sect. 2 through a combination of mechanisms. We will now give an overview of these mechanisms, see also Fig. 1 for a high-level overview of the BenchFoundry architecture.

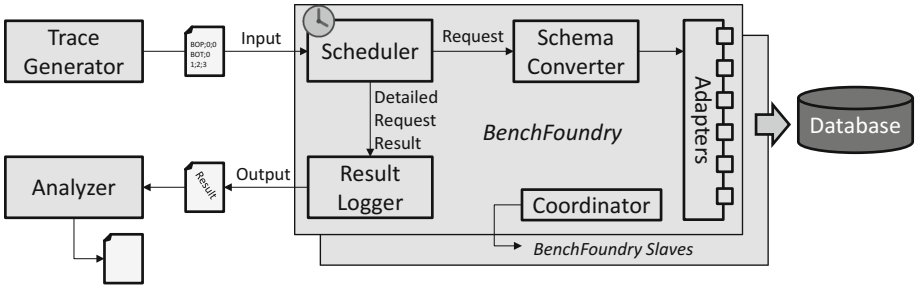


Fig. 1. High-level architecture

3.1 Trace-Based Workload Generation

The first novelty is that we break down the workload generator component into two components: a trace generator and a scheduler. The trace generator produces a workload trace which precisely specifies the order of operations and the time when each operation shall be executed relative to the experiment start. This trace is generated independently of a specific benchmark run, in fact, it may be based on real application traces and is supposed to be reused frequently. At runtime of the experiment, the scheduler retrieves entries from the trace and submits them as independent tasks to a variable-sized thread pool. This happens at the time specified in the trace – BenchFoundry also tracks scheduling precision.

Following this trace-based approach enables us to have fully deterministic executions where all elements of chance are captured within the trace generator (R7). Beyond repeatability, this trace-based approach also means that BenchFoundry is the first benchmarking toolkit where workloads become mere configuration files: Instead of writing a new workload generator, which typically includes aspects like thread management but also coordination in case of a distributed deployment, we can add new workloads to BenchFoundry by creating a static configuration file – manually, based on an existing real application trace, or programmatically through an existing or new trace generator. While one could argue

that writing a trace generator instead of a workload generator only shifts the problem, a workload generator has to be rewritten for every benchmark implementation while a trace generator has to be written just once. Hence, BenchFoundry is also extensible for new workloads (parts of R4).

3.2 Runtime Measurements and Offline Analysis

The second novelty is that we separate data collection from data interpretation: To our knowledge, existing general purpose database benchmarking tools all calculate metrics at runtime – obviously, this is not very extensible for new metrics. Furthermore, some measurement approaches require data from various measurement clients (e.g., [4]), i.e., calculating metrics creates a significant amount of communication. However, this is something to be avoided at runtime so as not to interfere with precise workload generation. In BenchFoundry, we log detailed results about every single request that we execute. At the moment, we log the operation ID (which together with the trace file specifies all details of the operation), start and end timestamps, returned values for reads, and whether the operation was successful. After completing the benchmark, these raw results are interpreted through offline analysis, i.e., we separate data collection from data interpretation.

Based on this information, calculating quality levels at arbitrary levels of aggregation is possible for a variety of system qualities and metrics, e.g., latency and throughput, consistency (staleness, ordering guarantees), or violations of ACID guarantees.

BenchFoundry logs results as detailed as possible (R6) and, thus, provides information for a variety of qualities and quality metrics (R1). It also aims to transform non-functional requirements into quality metrics (parts of R2).

3.3 Application-Focused Workload Abstraction

Existing benchmarking tools like YCSB typically use independent operations that are generated synthetically as a basis of their workload model; TPC benchmarks usually use transactions comprising multiple operations as their base unit but also describe the notion of emulated clients. In this regard, TPC benchmarks resemble real applications more closely: real database-application interactions typically happen within the scope of a session during which a sequence of transactions is executed by the storage service.

In BenchFoundry, we make this session explicit in our workload abstraction: The basic unit of execution is the *business process*[7]. A business process describes a sequence of database-application interactions, i.e., all interactions that would happen within the scope of a client session for real world applications. All entries of a business process are executed strictly sequentially, there is never parallelism.

[7] Which should not be confused with the process understanding of the BPM community.

The subunit of a business process is called *business transaction*. A business transaction is a logical sequence of *business operations* that should ideally, if supported by the storage service, be executed as ACID transactions. However, in the absence of transactional features, BenchFoundry simply executes these on a best effort base and tracks ACID violations. This allows us to compare transactional and non-transactional storage services fairly.

On a logical data schema level, a business operation is an atomic unit that corresponds to a database query. However, only RDBMS use a normalized data schema as their physical schema. Other database classes, e.g., column stores, rely on denormalization where data is kept redundantly to avoid costly queries. Logical updates may, hence, require several service calls. In BenchFoundry, we reflect this through the use of database class-specific *requests*, e.g., a column store request. In the case of RDBMS, each business operation has exactly one request; in the case of other database classes, one or more depending on the physical schema design.

All input files of BenchFoundry are specified on the logical schema level, i.e., BenchFoundry does not make assumptions on the physical schema of the storage service. Instead, it reads the logical schema, automatically creates a physical schema recommendation from this, and then creates the requests based on the physical schema and the original query. In case of column stores and key-values stores, we plan to use one of the approaches from [5,7] for this, for RDBMS we can simply use the normalized data schema, for other datastore classes schema mappings need to be determined and imported manually.

Using a workload abstraction that focuses on the behavior of client applications instead of taking the perspective of the storage service, is a very natural way of modeling workloads. Therefore, using the concepts of business processes, transactions, and operations easily allows developers to model application behavior which then results in the workload that the database experiences. The alternative of using independent operations as a base unit may also lead to very realistic workloads – however, we believe that this is much harder to "get right". As such, BenchFoundry (which is not a benchmark itself) does not guarantee R3 but certainly helps developers achieve it through an easy-to-use workload abstraction. By differentiating logical and physical schema levels, BenchFoundry also gets rid of functional requirements on the SUT which helps for a broad applicability (R2).

3.4 Managed Distribution and Benchmark Phases

BenchFoundry has been designed to be regularly deployed on multiple machines that together form a BenchFoundry cluster. As basic unit of distribution, we use business process instances, i.e., when we run BenchFoundry in a distributed setting, a trace splitter will assign each business process in the trace to a different BenchFoundry instance. As business processes are by definition independent (each process includes all interactions within the scope of a client session), these instances can be executed independently without requiring coordination.

For other aspects which require coordination, BenchFoundry follows a master-slave approach – however, the master cannot become a bottleneck for the system as all coordination happens before the actual benchmark run (see also Fig. 2):

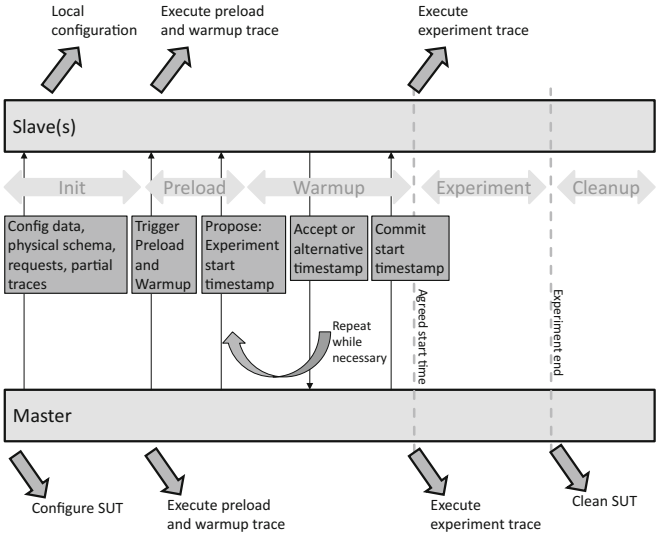


Fig. 2. Execution Phases and Distributed Coordination

During the init phase, the master parses all input files, splits the preload and experiment traces, and configures the SUT (e.g., by creating tables in an RDBMS). Afterwards, the master forwards the partial traces, the warmup trace, and configuration details (including physical schema and requests) to all slaves. When all BenchFoundry instances have been configured, the master signals all slaves to proceed to the preload phase during which the initial data set is loaded into the SUT. This is immediately followed by the warmup phase which serves to warm up database caches.

Once the warmup phase is started, the master proposes a start timestamp for the experiment phase to all slaves. For this, it uses a Two-Phase Commit variant: Instead of denying or accepting the proposal, slaves simply respond with an alternative (later) start timestamp or the proposed timestamp if it is accepted. The master then broadcasts a "commit" with the latest returned timestamp. At the agreed start time, all business processes of the warmup phase are forcibly terminated and the scheduler for the experiment phase is started. Instances that have completed their (partial) experiment trace, terminate autonomously and assert that all results have been logged. The master then proceeds to clean up the SUT, i.e., deletes all data that was written during the benchmark, etc.

All in all, BenchFoundry instances only communicate (a) for distribution of input data, (b) for starting the preload phase, and (c) for agreeing on the start

timestamp of the experiment phase. The trace and the business process-based workload abstraction already capture all dependencies in the workload which is why we use them as unit of distribution. Based on this, all other decisions can be made entirely locally without requiring communication. However, it is, therefore, necessary to synchronize the clocks of all BenchFoundry machines.

Since the BenchFoundry design avoids coordination where possible and keeps it outside of the experiment phase when unavoidable, we believe BenchFoundry to be highly scalable. As such, the system is also a natural fit for distributed or even geo-distributed deployments (R5). At the same time, using the master-slave approach together with the phase concept allows us to focus on ease-of-use: All slaves are only started with a port parameter, the master parses all input files and forwards it to slaves which self-configure upon receipt. The master also configures and cleans up the SUT.

4 BenchFoundry Implementation

In this section, we will give an overview of select implementation aspects of our proof-of-concept prototype[8]. We begin by discussing the input formats, before describing already implemented trace generators.

4.1 Input Formats

In BenchFoundry, we decided to split the input trace into several files: Especially long-running benchmarks will have many repetitive entries in the trace, e.g., when issuing an operation repeatedly with different parameters. We, hence, use deduplication both in the input files but also for the in-memory data structures which follow the same format. Figure 3 gives an overview of the trace input files.

Operation List: This file contains all queries that are used in a given workload along with a unique ID. In the queries, we use wildcards for the actual parameter values, e.g., the actual ID value in "SELECT * FROM customer WHERE id=?". All operations are kept in memory where queries are accessible by their ID. In the input file, we use SQL to specify the queries.

Parameter List: This file contains parameter sets along with a unique ID and is also kept in-memory. Using both a parameter ID and an operation ID, an executable query can be assembled at runtime.

Trace: This file contains information on business processes, their composition, and their respective start time. As the file will typically be very large, it contains all entries ordered by time and can, therefore, be read in a streaming mode with a lookahead buffer. Typically, a scheduler will read at least two seconds ahead in the trace to have sufficient time for parameter and operation lookups and, thus, to guarantee on time scheduling. The file format itself demarcates business processes with BOP/EOP (begin/end of process) and business transactions

[8] https://github.com/dbermbach/BenchFoundry.

within those with BOT/EOT (begin/end of transaction). The BOP entry also includes the (relative) start timestamp of the process whereas the BOT entry may include an optional delay before starting the respective transaction to model think time of emulated users. Operations in the main trace file are specified as a combination of operation ID, parameter ID, and custom parameter ID (see below). In a BenchFoundry deployment, we will typically have one trace each for preload, warmup, and experiment phase.

Custom Parameter List: This file uses the same format as the parameter list. However, these entries are not used by BenchFoundry directly. Essentially, custom parameters are parameters that are uninterpretedly passed to the actual storage service connectors which may (but do not have to) use them. Example use cases could be consistency levels or the IP address of a specific replica.

Other Files: Beyond the trace files, we also have an input file for the logical data schema which uses SQL DDL statements and a general properties file.

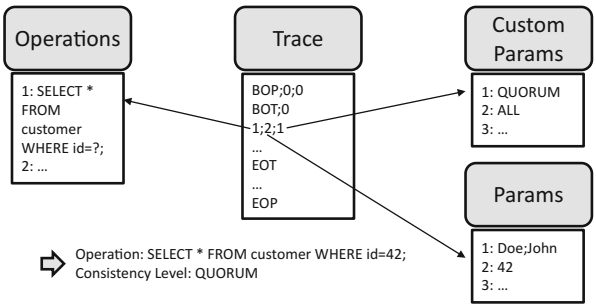


Fig. 3. File and In-Memory Representation of Workloads

When creating an experiment trace, only the experiment trace and the operation list are mandatory. Preload and warmup traces as well as custom parameters are optional and parameters may already be included in the queries.

4.2 Implemented Workloads

Currently, we have implemented two trace generators for BenchFoundry: The first generates traces based on the consistency benchmarking approach from [4] (consbench), i.e., it creates a workload that is designed to provoke upper bounds for staleness. The consbench trace generator is interactively configured with, e.g., the estimated number of replicas, the desired benchmark duration, and the number of tests. It then automatically decides on an appropriate number of BenchFoundry machines based on probability analysis as described in [4] and builds the corresponding input files. The second trace generator is based on

TPC-C[9], TPC's current order and inventory management benchmark. The original TPC-C benchmark describes four transactions; in our BenchFoundry trace generator, users can configure how they want to assemble these into processes.

To ease the implementation of additional trace generators, we have implemented an easy-to-use builder class where trace generators can simply create new business processes through method chaining. This builder class then automatically handles parameter and query deduplication while creating the correct input formats.

5 Evaluation

In this section, we present the results of our evaluation beyond the already presented proof-of-concept implementation; specifically, we present two things: First, we discuss how BenchFoundry fulfills the requirements described in Sect. 2. Second, we take a "systems perspective" and present the results of two experiments which show that BenchFoundry offers precise scheduling for normal load levels but is able to sustain a higher throughput level at the cost of accuracy as well as results showing that BenchFoundry can easily be scaled through distribution.

5.1 Discussion of Requirements in BenchFoundry

In this section, we will briefly discuss how BenchFoundry addresses each of the requirements for modern storage service benchmarks from Sect. 2.

(R1) Multi-Quality: R1 demands that benchmarks should measure all sides of a particular tradeoff. In BenchFoundry, we log detailed results about each operation including start and end timestamp as well as the values actually written. This allows to determine consistency behavior, performance, availability, and other qualities. Scalability and elasticity can be measured by varying the workload intensity (i.e., the "density" of business process starts in the workload trace).

(R2) No Assumptions: R2 demands that benchmarks should make as little assumptions on the SUT as possible, instead they should measure deviations from an ideal state. BenchFoundry only assumes that an SUT should expose a service interface with operations for data manipulation. The mapping to a concrete storage service is handled through adapter mechanisms.

(R3) Realistic Workloads: R3 demands that benchmarks should use realistic application-driven workload that mimick the target application as close as possible. BenchFoundry itself is not a benchmark but rather an execution environment for arbitrary application-driven benchmarks. For this purpose, BenchFoundry offers a partly-open workload model [15] based on business operations, business transactions, and business processes (which is the most realistic one for

[9] tpc.org/tpcc

most scenarios) to benchmark designers. It also comes with a scheduler for closed workload models as used in YCSB [8] and an open workload model is obviously a special case of the partly-open one for which the respective scheduler in BenchFoundry can be "misused". Therefore, we believe that BenchFoundry offers as much support for R3 as possible without actually designing a benchmark.

(R4) Extensibility: R4 demands that benchmarks should be extensible and configurable to account for both future application scenarios as well as new storage services. BenchFoundry executes arbitrary workload traces and is based on an adapter architecture for storage systems as presented in Fig. 1; it is also extensible with regards to quality metrics measured as it separates the benchmark run from data analysis and logs raw measurement results. We, hence, believe that it is safe to conclude that it fulfills R4.

(R5) Distribution: R5 demands that benchmarks should be designed for distribution. BenchFoundry uses a workload model that can easily be distributed and shifts all necessary coordination logic to a pre-benchmark phase.

(R6) Fine-Grained Results: R6 demands that benchmarks should always log fine-grained results. We could not think of any further measurement results that could possibly be logged in BenchFoundry. However, extending this would be straightforward.

(R7) Deterministic Execution: R7 demands that benchmarks should be able to deterministically re-execute the exact same workload. We address this by using a trace-based workload model which is fully deterministic.

(R8) Ease-of-use: R8 demands that benchmarks should have ease-of-use as a core focus. We tried to reach this goal as much as possible, e.g., by automatically configuring slave machines; if we managed to be successful is to be decided by BenchFoundry users.

5.2 Experiments

While we believe that BenchFoundry fulfills all the requirements initially identified, we also wanted to take a "systems perspective" and experimentally verify whether BenchFoundry is able to scale through distribution (the *DISTRIBUTION* experiment) and also to analyze how scheduling precision of workloads, i.e., the repeatability and determinism of workload execution, is affected by overloading the machines (the *LOAD* experiment).

Experiment Setup. For our experiment setup, we chose a setup that stresses BenchFoundry while keeping our SUT lightly loaded. In a regular benchmarking experiment, this would of course be exactly the other way around. We, therefore, deployed up to five BenchFoundry instances on Amazon EC2[10] t2.small instances and a single MariaDB node as SUT on an m4.xlarge instance.

[10] aws.amazon.com/ec2.

We preloaded the database with a small data set of 4211 rows in 9 tables based on the TPC-C specification. For our workload, we also used TPC-C as a basis and designed 4 different business processes with one of the TPC-C transactions each as business transaction; transactions always contained several business operations. We configured our trace generator so that it created a trace with a base unit of 2 business processes per second (constant target throughput) that could be scaled through a load factor. In the following, we will refer to throughput based on the load factor, e.g., a load factor of 10 means that we ran a workload that scheduled 20 business processes per second, each containing a single business transaction with several business operations. In each test run, we sustained the respective throughput for 120 s.

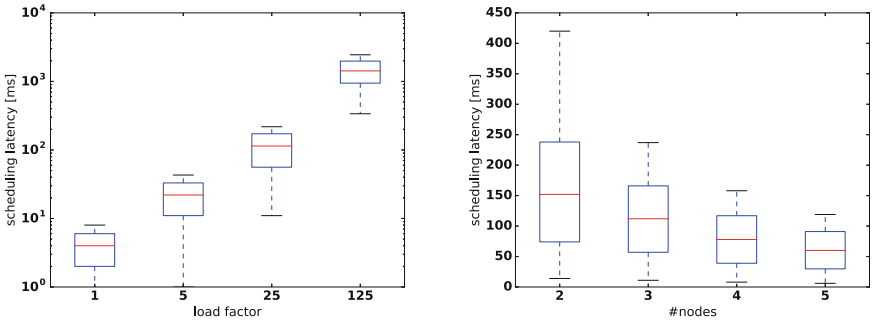
As a metric for the scheduling precision and, thus, the ability to precisely re-execute a given workload, we used the scheduling latency which is defined as the absolute difference in time between the planned start timestamp of a business process and its actual start timestamp. We would also like to point out that collecting data for this metric along with debug-level logging, of course, negatively affects the scheduling latency, i.e., users can expect values at least as good in real benchmark runs.

As already mentioned, we ran two experiments: the *LOAD* experiment and the *DISTRIBUTION* experiment. In the *LOAD* experiment, we used a single BenchFoundry instance and measured the scheduling latency for different target throughputs to (a) analyse scheduling precision for normal load levels and (b) to measure maximum sustainable throughputs on a single instance. We, therefore, tried to use the load factors 1, 5, 25, 125, 250 and 625. In the *DISTRIBUTION* experiment, we used a constant load factor of 50 (a level that was no longer sustainable on a single small instance with reasonable scheduling precision) and ran that workload distributed over two to five BenchFoundry instances.

Results. For each experiment, we show a single chart with a single boxplot for each run. Each boxplot represents a total of 6,000 measurements and shows 5, 25, 50, 75, and 95 percentiles (of scheduling latency in ms) for the corresponding test run.

In the *LOAD* experiment (see Fig. 4a, note the logarithmic scale), we were not able to reach load factors of 250 or 625. In both cases, we encountered an out of memory error so that we recommend to always pay special attention to heap size configuration. In all other experiment runs, we saw the expected behavior: low scheduling latencies for normal load levels that increased with higher sustained throughputs. At a load level of 125, the (small) instance was effectively overloaded resulting in unacceptably high scheduling latencies.

In the *DISTRIBUTION* experiment (see Fig. 4b), we also saw the expected results: BenchFoundry scales almost linearly with the number of nodes, i.e., increasing the number of nodes improves scheduling precision for constant workloads. Since BenchFoundry instances are completely independent during benchmark runs, doubling the load while using twice the number of machines should



(a) *LOAD* Experiment (b) *DISTRIBUTION* Experiment

Fig. 4. Experiment Results

not affect scheduling precision negatively, thus, also guaranteeing linear scalability in this regard.

All in all, BenchFoundry is – as expected – able to offer a high scheduling precision (<5–10 ms) and, thus, repeatability for workloads at "normal" load levels, i.e., when the machine is not fully loaded, and experiments indicate that it scales well.

6 Limitations and Effects of Design Choices

In a distributed trace, there may be situations where a faster database service or more powerful compute instances running BenchFoundry may allow some clients to complete a process more quickly than others. This can, of course, result in out-of-order execution of operations from different processes that have implicit dependencies. However, such implicit dependencies should be avoided in a workload design following our process abstraction. Furthermore, faster or slower execution of businesses processes may endanger precise repeatability for very long process instances. Future extensions of BenchFoundry could replace fixed delays between transactions with dynamic delays that depend on execution speed, e.g., execute at $t = 100$ or after 50 ms whatever happens first.

BenchFoundry needs to log fine-grained results to satisfy (R6) which leads to a certain overhead. In our design, we aimed to mitigate any potential impacts. First, we decoupled logging from execution and measurements by using two separate modules for this which communicate asynchronously. Second, we avoid network contention and minimize CPU overheads by having BenchFoundry instances only log raw data locally: Costly correlation of measurements and interpretation of raw results is done after completion of the benchmark run. Third, BenchFoundry was designed to scale well so that the overhead of writing fine-grained results instead of coarse aggregation can be mitigated by adding additional virtual machines to the benchmarking cluster. We believe that this keeps any impact on measurement and workload execution within reasonable

bounds. The cost for additional machines is the price we have to pay for getting more meaningful results – in today's inexpensive compute services, this should be negligible in most cases.

7 Conclusion

In this paper, we have presented BenchFoundry, a benchmarking framework that can execute arbitrary application-driven workloads in a distributed deployment while measuring multiple system qualities of a cloud storage service. To our knowledge, BenchFoundry is the first framework that uses trace-based workloads where workloads become mere configuration files for this purpose. Beyond this convenience aspect, trace-based workloads also guarantee precise repeatability of benchmark runs.

We started by identifying requirements for modern storage benchmarks. Based on this, we presented the design and architecture of BenchFoundry before covering implementation details and evaluating our approach. In future work, we plan to implement additional trace generators and database connectors.

Acknowledgements. We would like to thank Sherif Sakr for his contributions during the early stages of the project, Daniel Wenzel for his support during some of our experiments, and Amazon Web Services for providing free access to their services.

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design: cap is only part of the story. IEEE Comput. **45**(2), 37–42 (2012)
2. Anderson, E., Li, X., Shah, M.A., Tucek, J., Wylie, J.J.: What consistency does your key-value store actually provide? In: Proceedings of HOTDEP. USENIX (2010)
3. Bermbach, D., Kuhlenkamp, J., Dey, A., Sakr, S., Nambiar, R.: Towards an extensible middleware for database benchmarking. In: Nambiar, R., Poess, M. (eds.) TPCTC 2014. LNCS, vol. 8904, pp. 82–96. Springer, Cham (2015). doi:10.1007/978-3-319-15350-6_6
4. Bermbach, D.: Benchmarking Eventually Consistent Distributed Storage Systems. Ph.D. thesis, Karlsruhe Institute of Technology (2014)
5. Bermbach, D., Mueller, S., Eberhardt, J., Tai, S.: Informed schema design for column store-based database services. In: Proceedings of SOCA. IEEE (2015)
6. Bermbach, D., Wittern, E., Tai, S.: Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective. Springer, Cham (2017)
7. Chebotko, A., Kashlev, A., Lu, S.: A big data modeling methodology for apache cassandra. In: Proceedings of BigData. IEEE (2015)
8. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of SOCC. ACM (2010)
9. Difallah, D.E., Pavlo, A., Curino, C., Cudre-Mauroux, P.: OLTP-bench: An extensible testbed for benchmarking relational databases. Proceedings of VLDB **7**(4), 277–288 (2013)

10. Folkerts, E., Alexandrov, A., Sachs, K., Iosup, A., Markl, V., Tosun, C.: Benchmarking in the cloud: what it should, can, and cannot be. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 173–188. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36727-4_12
11. Huppler, K.: The art of building a good benchmark. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 18–30. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10424-4_3
12. von Kistowski, J., Arnold, J.A., Huppler, K., Lange, K.D., Henning, J.L., Cao, P.: How to build a benchmark. In: Proceedings of ICPE (2015)
13. Müller, S., Bermbach, D., Tai, S., Pallas, F.: Benchmarking the performance impact of transport layer security in cloud database systems. In: Proceedings of IC2E. IEEE (2014)
14. 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. In: Proceedings of SOCC. ACM (2011)
15. Schroeder, B., Wierman, A., Harchol-Balter, M.: Open versus closed: a cautionary tale. In: Proceedings of NSDI, vol. 6, p. 18 (2006)
16. Seybold, D., Domaschka, J.: A cloud-centric survey on distributed database evaluation. In: Proceedings of ADBIS (2017)
17. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. In: Proceedings of CIDR (2011)