BenchFoundry: A Benchmarking Framework for Cloud Database Services

David Bermbach and Jörn Kuhlenkamp
Information Systems Engineering Research Group
TU Berlin
Berlin, Germany
Email: {db,jk}@ise.tu-berlin.de

Akon Dey
Awake Networks Inc.
Mountain View, California, USA
Email: akon@awakenetworks.com

***Abstract*—Understanding quality of cloud database systems and services is often crucial. However, current cloud database benchmarks either come without an implementation, have strong disadvantages, or can only be used with relational database systems and services.**

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 OLTP database system or 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.

1. Introduction

Today, the sheer number of available cloud storage services and database systems is staggering – in August 2016, nosql-databases.org lists more than 200 NoSQL database projects, a number that does not even include relational database systems and services (RDBMS). Selecting a system or service from this extensive set for an application scenario is done based on three criteria: (a) functionality, i.e., implemented features, data model, etc., (b) non-functional properties, i.e., the system qualities provided by the system or service, and (c) trust aspects, i.e., whether the application developers believe that the system will be maintained or the service will continue to be available over the application lifetime. In this paper, we will focus on the comparability of systems and services in terms of quality; benchmarking provides the necessary insights into this.

While there is a plethora of previous work on database benchmarking, existing approaches have severe disadvantages or are too limited in their applicability: Some approaches, e.g., TPC benchmarks or OLTPBench [1], have strict functional and non-functional requirements on supported database systems and services which todayare only fulfilled by RDBMS. As such, these benchmarks cannot be used to study NoSQL systems and are often only a specification and not actually implemented. Other approaches, e.g., YCSB [2] or YCSB++ [3] are essentially micro-benchmarks. While these are useful for understanding how tiny changes in workloads affect system quality or for testing isolated database features, they are not a good fit for use cases such as selecting or optimizing the configuration of a database system since micro-benchmarks rarely mimick application workloads realistically. Other criteria where existing approaches are lacking are aspects like extensibility in terms of workloads, multi-quality measurements, (geo-)distribution support of the benchmark out of the box, fine-grained result collection, or ease-of-use. We will discuss requirements for modern benchmarks in detail and give an overview of related work in section 2.

Therefore, application developers currently have two options: First, design their own benchmark or adapt an existing one. Obviously, this means that they also have to implement the benchmark for all database systems of interest from scratch which is particularly challenging when scalability matters. Second, use existing tools that are inherently limited in their applicability or the meaningfulness of produced measurement results.

Focusing on the first option, we present in this paper the result of designing and actually implementing ideas from our previous vision paper [4]: 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 OLTP database system or 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.

This paper is structured as follows: In section 2, we discuss in detail requirements for modern benchmarks along with related work, thereby, again motivating our contributions. Afterwards, in sections 3 and 4 we present BenchFoundry, starting with a high-level overview before going into implementation details. Finally, we evaluate BenchFoundry through select experiments (section 5) before concluding in section 6.

2. Motivations and Related Work

In this section, we will identify requirements for modern database benchmarks and their implementations. We will also discuss existing work in this field.

Traditionally, database benchmarking has mainly been done for performance, e.g., through TPC[1] benchmarks or with YCSB [2]. Over the last few years, some approaches have been developed for consistency benchmarking with varying degrees of meaningfulness[2], e.g., [3], [8]–[15], security impacts on performance [16], as well as an open source project for testing ACID isolation guarantees[3]. 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 [17], [18] – being top ranked for one quality is trivial when disregarding the respective other qualities. As an analogy, sports cars and hybrid cars obviously solve the tradeoff between fuel efficiency and top speed differently which makes them each a better or worse fit depending on the user's priorities. To make such tradeoff decisions transparent, modern benchmarking should always imply multi-quality benchmarking.

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

Existing benchmark tools often have strict functional and non-functional requirements on supported database systems and services, e.g., requiring transactional features with strict ACID guarantees or data models which today are only offered by RDBMS [1]. However, it would be preferable to reach a broader applicability and stronger portability [6], [7] by transforming such strict requirements into measured qualities instead. For instance, transactions could also be executed in a best-effort way while tracking isolation violations as an additional quality metric.

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

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 are typically preferable over synthetic micro-benchmarks like YCSB. However, 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. At the same time, existing application-driven benchmarks, e.g., the ones developed by TPC, have other shortcomings. They often come without an implementation leaving the interested user to the repetitive and tedious task of reimplementing the specification for every database system or service.

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

Modern applications evolve at a 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 database systems and 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 [2], [9]. 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 RDBMS, document stores, or key value stores. We believe that a modern benchmark should distinguish a logical and physical data model in its adapter layer. Having little assumptions regarding the SUT also helps for this requirement.

> **(R4) Extensibility:** *Benchmarks should be extensible and configurable to account for both future application scenarios as well as new data storage solutions.*

Modern applications as well as the underlying storage layer, no matter if self-hosted system or cloud service, are also inherently distributed if not 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 at will. Parallelization through distribution is also important when measuring the scalability of database systems and services or simply for benchmarking a system that is already at scale (scalability of the benchmark tool). Also, some benchmarking approaches heavily rely on distributed execution, e.g., [9]. However, distributing workloads is a challenging problem, e.g., making sure that an insert was completed before reading or updating the same key. As another example, YCSB has frequently been used for evaluation sections in research papers. Paper evaluations that are based on running multiple YCSB machines concurrently that each execute a workload with a non-uniform distribution for selecting keys, are partially flawed: Adding two or more independently generated single peak distributions on top of each other will only by chance result in another single peak distribution. Therefore, coordination in distributed benchmarking is crucial, YCSB++ [3] is supposed to fix this for YCSB[4].

> **(R5) Distribution:** *Benchmarks should be always be distribution-aware and implementations should*

1. tpc.org
2. 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 [4]–[7].
3. github.com/ept/hermitage
4. To which degree is not specified in the paper and the source code seems not to be available anymore.

come with the necessary coordination logic for running multiple instances in parallel.

Often, benchmarking tools only report aggregated results, e.g., [2]. 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 [9], [10] would never have been found if only aggregates or even results in the form of a CDF had been available. Benchmarking tools should, hence, log detailed results at operation level, i.e., for each operation the outcome, start and end timestamp, retrieved results for read queries, etc. Of course, this causes an extra load on the benchmarking machine – another argument in favor of distributing the benchmark to avoid the measurement machine becoming a bottleneck.

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

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 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 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. One of the arguments in favor of a randomized workload creation is to avoid situations where database vendors optimize for a specific workload – we believe that the random component should be used when generating the trace.

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

A benchmark should focus on ease-of-use to foster adoption and use. Often, it is not possible to benchmark all systems and 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 which includes a simple way to setup the benchmark tool as well as documentation but also possible help for setting up the SUT. On the other hand, this obviously requires benchmarks to also come with an implementation as done for the more recent TPC benchmarks.

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

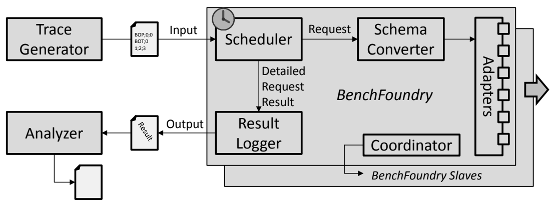

Figure 1. High-Level Architecture

3. BenchFoundry Design and Architecture

In BenchFoundry, we address each of the requirements from section 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.

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 specifies precisely the order of operations and the time when each operation shall be executed[5]. This trace is generated independently of a specific benchmark run, in fact, it may be based on real application traces and may 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 a trace generator. 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 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., [9]), i.e., calculating metrics creates a significant amount of communication. However, this

5. In BenchFoundry, we use a relative time notion for all components by counting milliseconds before and after the predetermined experiment start time.

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:

- *Performance:* Latency and throughput.
- *Availability:* Success rates by operation type, mean time between failures, and mean time to repair.
- *Consistency:* Staleness (t-Visibility, k-Staleness [18], [19]) and Ordering (violations of monotonic reads, monotonic writes, read your writes [18], [20]), probably also k-Atomicity, Δ-Atomicity [13], and related metrics.
- *Others:* Violations of ACID guarantees[6].

For now, we could not imagine any other information that should be logged – still, extending this at a later time is straightforward. Therefore, BenchFoundry already logs results as detailed as possible (R6) and provides, thus, 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 database.

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 (and is, thus, always linear) 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 and there are, unless specified, no artificial delays.

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 database system, be executed as transactions with full ACID guarantees. However, in the absence of transactional features, BenchFoundry simply executes business transactions on a best effort base and tracks ACID violations. This allows us to compare transactional and non-transactional database systems and services fairly; see also fig. 2.

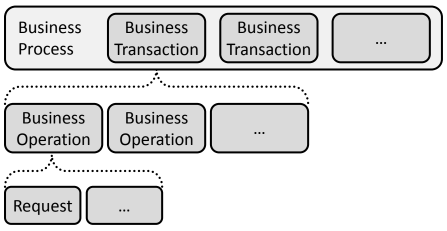

Figure 2. Workload Abstraction

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, i.e., typically each entity has its own relational table. Other database classes like column stores or key-value stores rely on denormalization, i.e., data is kept redundantly to avoid costly queries. 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 database. Instead, we read the logical schema, automatically create a physical schema recommendation from this, and then create the requests based on the physical schema and the original query. In case of column stores and key-values stores, we use the approach from [21], 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 database system or 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 R3[8].

By differentiating logical and physical schema levels, BenchFoundry also gets rid of functional requirements on the SUT which helps for a broad applicability (R2).

6. We will describe our notion of *business transactions* which is necessary for this in section 3.3.

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

8. During our developments, we have seen that this workload abstraction can also be misused for representing micro-benchmark workloads.

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 experiment is executed so that only the initial phases (see also fig. 3) may take longer to complete:

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 2PC 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 sends a "commit" with the latest returned timestamp to all slaves. When the start timestamp is reached, 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 business process instances 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, e.g., based on NTP.

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

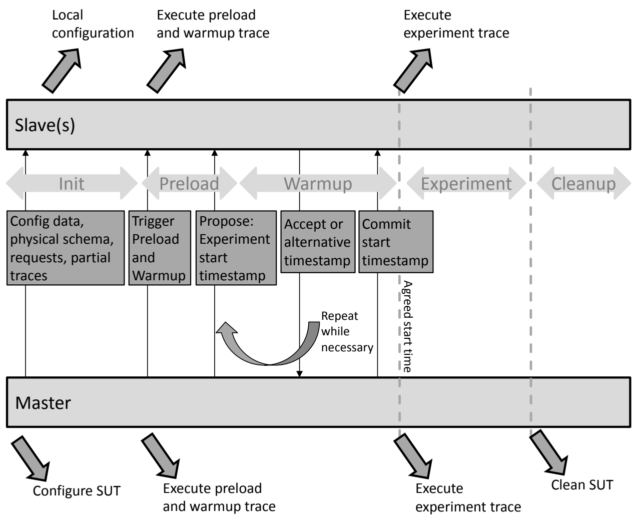
Local configuration
Execute preload and warmup trace
Execute experiment trace
Slave(s)
Init
Preload
Warmup
Experiment
Cleanup
Config data, physical schema, requests, partial traces
Trigger Preload and Warmup
Propose: Experiment start timestamp
Accept or alternative timestamp
Commit start timestamp
Agreed start time
Repeat while necessary
Master
Configure SUT
Execute preload and warmup trace
Execute experiment trace
Clean SUT

Figure 3. Execution Phases and Distributed Coordination

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 – we are continuing to extend this, e.g., towards VM provisioning, test automation [22], etc. to further improve ease-of-use (R8).

4. BenchFoundry Implementation

In this section, we will give an overview of select implementation aspects of our proof-of-concept prototype which we have not yet covered in section 3 and will also describe the current state of our implementation. We will start by giving an overview of the input and output formats of BenchFoundry, before describing the trace generators that we have already implemented. Afterwards, we will give a brief overview of how we deal with conversions between the logical and physical schema layers before describing the current state of our prototype.

Note to the reviewers: We are actively developing BenchFoundry, i.e., we will probably have to rewrite large parts of the subsections that refer to the current state/current limitations for a camera-ready version. The source code will be available online in the next few days.

4.1. Input and Output 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., because the same operations are issued repeatedly with different parameters. We, therefore, use deduplication both in the input file formats but also for the in-memory data structures which follow the same format. Figure 4 gives an overview of the trace input files.

Operation List: The operation list contains all queries that are used in a given workload along with a unique ID. In

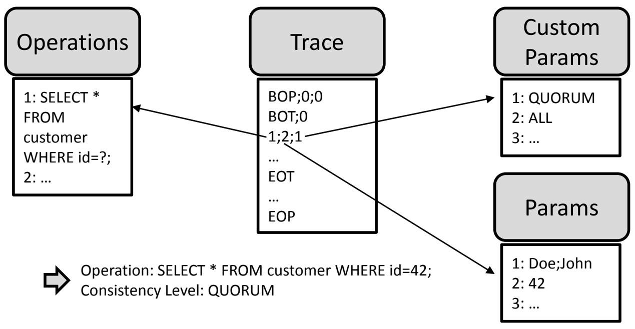
Operations
1: SELECT * FROM customer WHERE id=?;
2: …
Trace
BOP;0;0
BOT;0
1;2;1
…
EOT
…
EOP
Custom Params
1: QUORUM
2: ALL
3: …
Params
1: Doe;John
2: 42
3: …
Operation: SELECT * FROM customer WHERE id=42;
Consistency Level: QUORUM

Figure 4. File and In-Memory Representation of Workloads

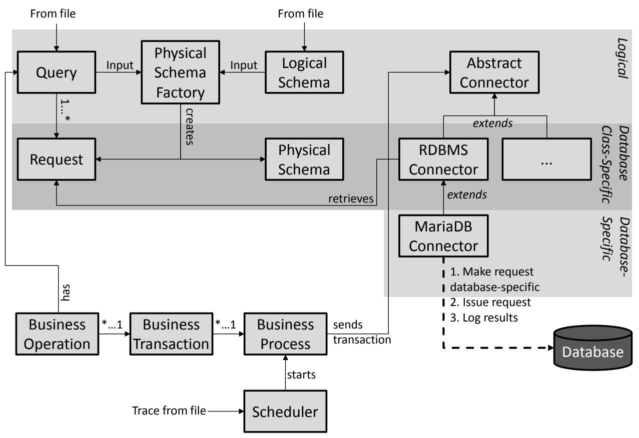

Figure 5. Overview of Components Handling Schema Conversions

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: The parameter list 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: The main trace 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 – with standard settings 3.5 seconds – 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 and business transactions within those with BOT/EOT. 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 times 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 – INSERT statements only for the preload trace, SELECT queries only for the warmup trace, and the actual experiment workload in the experiment trace.

Custom Parameter List: The custom parameter list 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 database connectors which may (but do not have to) use them. Example use cases could be consistency levels (e.g., QUORUM, ALL, or ONE in Cassandra [23]) 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. For all input files as well as the result log we use the CSV format wherever possible.

When creating an experiment trace, there are only two mandatory input files: the experiment trace and the operation list. Preload and warmup traces are optional, parameters may already be included in the queries, and custom parameters can be omitted.

4.2. Implemented Workloads

Currently, we have implemented two trace generators for BenchFoundry:

The first generates traces based on the consistency benchmarking approach from [9], 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 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.

4.3. From Logical to Physical Schema

As already described in section 3.3, we dinstiguish logical and physical data schemas in BenchFoundry. However, we also have an intermediate, database class-specific schema. In this section, we will describe how BenchFoundry transforms input data into the physical schema mapping.

9. tpc.org/tpcc

As can be seen in fig. 5, we have a three layer class hierarchy for database connectors – one on the logical level, one on the database class-specific level, and one on the physical level. Each of them requires queries in a different format. Therefore, we use the abstract factory pattern for schema generation: concrete subclasses for the database class-specific level, e.g., a column store physical schema factory, are used to determine a good physical schema mapping, e.g., through denormalizing in the case of column stores. Based on this (semi-)physical schema, the respective factory then creates a sequence of request objects for each query. Again, we have classes like RelationalRequest or ColumnStoreRequest which are at this database class-specific level.

At runtime, when the scheduler starts execution of a business process, BenchFoundry retrieves the request sequence for each business operation and passes it to the concrete connector instance. Through OOP mechanisms, the database connector then transforms the abstract request first into the intermediate database class-specific version before making datastore-specific changes so that the requests can actually be executed against the database system or service.

4.4. Current Implementation State

BenchFoundry currently comprises about 12,000 lines of Java 8 code and another 4,000 comment lines; the open-sourced code can be found at[10]. In BenchFoundry, we use Thrift 0.9.3[11] for communication between BenchFoundry instances, log4j 2.6.2 for logging, JDBC for connecting to RDBMS, and JSQLParser 0.9.4[12] for parsing SQL DDL and DML statements.

In each area where BenchFoundry can be extended easily, we have implemented at least one proof-of-concept extension:

Database support: We have currently implemented a connector for MariaDB[13] which we also used for our evaluation; adding further RDBMS connectors is straightforward. We have also reimplemented most of the schema generation code for column stores and key-value stores as described in [21]; adding the missing pieces will be our next steps in this area.

Result analysis: We have currently implemented an analyzer which parses our BenchFoundry output files and calculates detailed performance metrics; we are currently working on a consistency analyzer that calculates t-Visibility and k-Staleness but also violations of monotonic reads, monotonic writes, and read your writes.

Workload generation: We have already discussed the existing trace generators in section 4.2.

All other functionality has been implemented and tested. Beyond what we have described above, BenchFoundry also comes with two different schedulers: The default scheduler for running open workloads and a second scheduler for executing closed workloads [24] – we use the second scheduler in preload and warmup phases.

10. github.com/dbermbach/BenchFoundry
11. thrift.apache.org
12. github.com/JSQLParser/JSqlParser
13. mariadb.org

5. Evaluation

In this section, we present the results of our evaluation beyond the already presented proof-of-concept implementation; specifically, we demonstrate two things: First, BenchFoundry offers precise scheduling for normal load levels but is able to sustain a higher throughput level at the cost of accuracy. Second, BenchFoundry can easily be scaled through distribution.

5.1. 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[14] t2.small instances and a single MariaDB node on an m4.xlarge instance. All instances were set up with 100 GB high-performance SSD disk space that had 5,000 provisioned I/O operations per second (io1, 5,0000 PIOPS) to avoid any bottlenecks on the EC2 disk I/O.

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 seconds.

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

14. aws.amazon.com/ec2

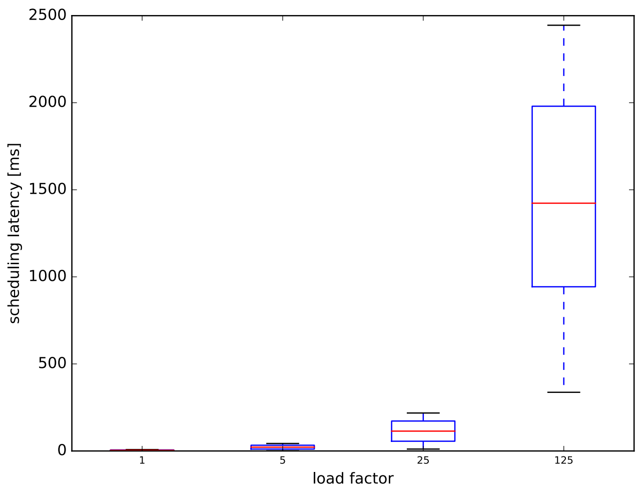

Figure 6. *LOAD* Experiment: BenchFoundry shows good scheduling precision for normal load levels and sustains higher throughputs at the cost of accuracy.

throughputs on a single instance. We, therefore, used the load factors 1, 5, 25, 125, and 625.

In the *DISTRIBUTION* experiment, we used a constant load factor of 50 (a level that, as we will see, was no longer sustainable on small instances with reasonable scheduling precision) and ran that workload distributed over 1, 2, 3, 4, and 5 BenchFoundry instances.

5.2. 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 for the corresponding test run.

In the *LOAD* experiment (see fig. 6), we were not able to reach load factors of 250 or 625. In both cases, we encountered an out of memory error which was probably caused by us using the default memory settings of the Java Virtual Machine. We, therefore, highly recommend explicitly setting a larger heap size for actual benchmark runs. Then, we would expect to be able to also sustain such high throughput levels. 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 instance was effectively overloaded.

In the *DISTRIBUTION* experiment (see fig. 7), we also saw the expected results: BenchFoundry scales almost linearly with the number of nodes, i.e., twice the number of nodes can sustain the same throughput level at approximately twice the scheduling precision[15]. Based on this but also based on our initial design with minimized interaction of instances, we believe it, thus, safe to say that BenchFoundry scales with the number of instances.

15. Obviously, this happens within bounds: we do not expect to find a linear correlation at very low or very high levels of sustained throughput.

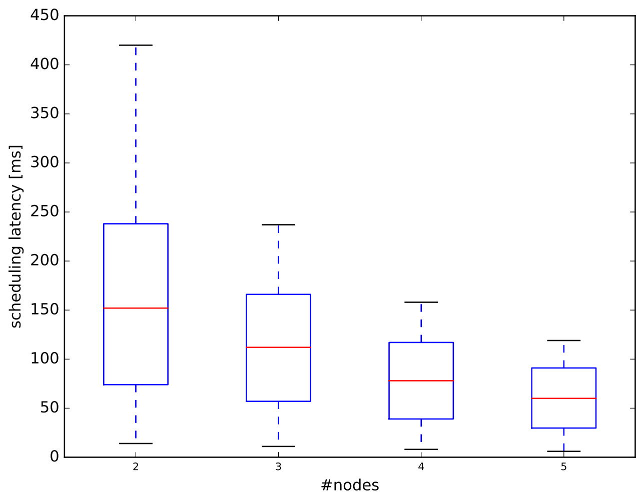

Figure 7. *DISTRIBUTION* Experiment: Adding more BenchFoundry nodes under constant throughput improves scheduling precision almost linearly.

5.3. Discussion

As expected, BenchFoundry is able to offer a high scheduling precision and, thus, repeatability for workloads at “normal” load levels, i.e., when the machine is not fully loaded. We also did see that open workload generation [24] with a variable number of threads puts a much higher strain on the machine's resources than a closed workload with a fixed size thread pool. We believe this to be due to the thread scheduling overheads already discussed in SEDA [25]. We further aggravated this issue by having single transaction processes which increases the number of necessary threads even more. However, in the era of inexpensive cloud resources, such an inefficiency in resource usage can easily be countered through additional compute resources if the benchmark implementation is scalable, i.e., the benchmark run becomes a bit more expensive in exchange for the obvious benefits of a trace-based scheduler, whereas this inefficiency would have been a showstopper in the pre-cloud era. We, therefore, believe that it is safe to say that BenchFoundry does what is was designed to do.

Regarding the other features and contributions that we claimed, these are hard or impossible to evaluate experimentally. For instance, aspects like fine-grained results (R6), extensibility (R4) or multi-quality (R1) are either implemented/supported or not – in the case of BenchFoundry this can easily be verified with a look at the publicly available source code.

6. 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 qualities of an OLTP database system or service. To our knowledge, BenchFoundry is the first framework that uses trace-based workloads, i.e., workloads become mere configuration files instead of code. Beyond this convenience aspect, trace-based workloads also guarantee precise repeatability of benchmark runs which we have also seen in our evaluation.

We started by identifying requirements based on literature and discussed strengths and weaknesses of related approaches. Based on this, we presented the design and architecture of BenchFoundry before covering select implementation details and evaluating scheduling precision, i.e., benchmark repeatability, as well as scalability of BenchFoundry through experiments.

In our next steps, we will mainly work on additional trace generators and database connectors. However, we will also try to fine-tune BenchFoundry further and update the source code to reflect lessons learned from our experiment preparations and experiments.

Finally, we would like to invite the research community to contribute to and to actively use BenchFoundry – we will be glad to offer support and guidance.

Acknowledgements

We would like to thank Sherif Sakr for his contributions during the early stages of the project as well as Stefan Tai for providing feedback on our paper. We would also like to thank Daniel Wenzel for his support during some of our experiments. Finally, we would like to thank Amazon Web Services who supported our experiments through grants.

References

[1] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltpbench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, 2013.

[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st Symposium on Cloud Computing (SOCC)*, ser. SOCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[3] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores," in *Proceedings of the 2nd Symposium on Cloud Computing (SOCC)*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 9:1–9:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038925

[4] D. Bermbach, J. Kuhlenkamp, A. Dey, S. Sakr, and R. Nambiar, "Towards an Extensible Middleware for Database Benchmarking," in *TPCTC 2014*. Springer, 2014, pp. 82–96.

[5] K. Huppler, "The art of building a good benchmark," in *Proceedings of the First TPC Technology Conference (TPCTC 2009)*. Springer, 2009, pp. 18–30.

[6] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, "How to build a benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE 2015)*. ACM, 2015, pp. 333–336.

[7] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun, "Benchmarking in the cloud: What it should, can, and cannot be," in *Proceedings of the 4th TPC Technology Conference (TPCTC 2012)*. Springer, 2013, pp. 173–188.

[8] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective," in *Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR)*, January 2011, pp. 134–143.

[9] D. Bermbach and S. Tai, "Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior," in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (MW4SOC)*, ser. MW4SOC '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:6. [Online]. Available: http://doi.acm.org/10.1145/2093185.2093186

[10] ——, "Benchmarking eventual consistency: Lessons learned from long-term experimental studies," in *Proceedings of the 2nd International Conference on Cloud Engineering (IC2E), to appear*. IEEE, 2014.

[11] D. Bermbach, L. Zhao, and S. Sakr, "Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services," in *Performance Characterization and Benchmarking*, ser. Lecture Notes in Computer Science, R. Nambiar and M. Poess, Eds. Springer International Publishing, 2014, vol. 8391, pp. 32–47. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-04936-6_3

[12] K. Zellag and B. Kemme, "How consistent is your cloud application?" in *Proceedings of the 3rd Symposium on Cloud Computing (SOCC)*, ser. SOCC '12. New York, NY, USA: ACM, 2012, pp. 6:1–6:14. [Online]. Available: http://doi.acm.org/10.1145/2391229.2391235

[13] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie, "Toward a principled framework for benchmarking consistency," in *Proceedings of the 8th Conference on Hot Topics in System Dependability (HOTDEP)*, ser. HotDep'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387858.2387866

[14] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie, "What consistency does your key-value store actually provide?" in *Proceedings of the 6th Workshop on Hot Topics in System Dependability (HOTDEP)*, ser. HotDep'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924908.1924919

[15] W. Golab, X. Li, and M. A. Shah, "Analyzing consistency properties for fun and profit," in *Proceedings of the 30th Symposium on Principles of Distributed Computing (PODC)*, ser. PODC '11. New York, NY, USA: ACM, 2011, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/1993806.1993834

[16] S. Müller, D. Bermbach, S. Tai, and F. Pallas, "Benchmarking the performance impact of transport layer security in cloud database systems," in *Proceedings of the 2nd International Conference on Cloud Engineering (IC2E), to appear*. IEEE, 2014.

[17] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *IEEE Computer*, vol. 45, no. 2, pp. 37–42, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1109/MC.2012.33

[18] D. Bermbach, "Benchmarking eventually consistent distributed storage systems," Ph.D. dissertation, Karlsruhe Institute of Technology, 2014.

[19] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 776–787, Apr. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2212351.2212359

[20] D. Bermbach and J. Kuhlenkamp, "Consistency in distributed storage systems: An overview of models, metrics and measurement approaches," in *Proceedings of the International Conference on Networked Systems (NETYS)*. Springer, 2013, pp. 175–189.

[21] D. Bermbach, S. Mueller, J. Eberhardt, and S. Tai, "Informed schema design for column store-based database services," in *Proceedings of the 8th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2015)*. IEEE, 2015.

[22] M. Klems, D. Bermbach, and R. Weinert, "A runtime quality measurement framework for cloud database service systems," in *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (QUATIC)*, Sept 2012, pp. 38–46.

[23] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” *SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[24] B. Schroeder, A. Wierman, and M. Harchol-Balter, “Open versus closed: A cautionary tale,” in *Proceedings of NSDI*, vol. 6, 2006, pp. 18–18.

[25] M. Welsh, D. Culler, and E. Brewer, “Seda: An architecture for well-conditioned, scalable internet services,” in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, ser. SOSP ’01. New York, NY, USA: ACM, 2001, pp. 230–243. [Online]. Available: http://doi.acm.org/10.1145/502034.502057