

A generic database benchmarking service

M. Kaufmann, Peter M. Fischer, D. Kossmann, N. May

Angaben zur Veröffentlichung / Publication details:

Kaufmann, M., Peter M. Fischer, D. Kossmann, and N. May. 2013. "A generic database benchmarking service." In 2013 IEEE 29th International Conference on Data Engineering (ICDE), 8-12 April 2013, Brisbane, QLD, Australia, edited by Christian S. Jensen, Chris Jermaine, Jiaheng Lu, Egemen Tanin, and Xiaofang Zhou, 1276-79. Piscataway, NJ: IEEE. <https://doi.org/10.1109/icde.2013.6544923>.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>



A Generic Database Benchmarking Service

Martin Kaufmann ^{#*1}, Peter M. Fischer^{†2}, Donald Kossmann ^{#3}, Norman May ^{*4}

[#]*Systems Group, ETH Zürich, Switzerland*

¹`martin.kaufmann@inf.ethz.ch`, ³`donald.kossmann@inf.ethz.ch`

[†]*Albert-Ludwigs-Universität Freiburg, Germany*

²`peter.fischer@informatik.uni-freiburg.de`

^{*}*SAP AG, Walldorf, Germany*

⁴`norman.may@sap.com`

Abstract—Benchmarks are widely applied for the development and optimization of database systems. Standard benchmarks such as TPC-C and TPC-H provide a way of comparing the performance of different systems. In addition, micro benchmarks can be exploited to test a specific behavior of a system.

Yet, despite all the benefits that can be derived from benchmark results, the effort of implementing and executing benchmarks remains prohibitive: Database systems need to be set up, a large number of artifacts such as data generators and queries need to be managed and complex, time-consuming operations have to be orchestrated. In this demo, we introduce a generic benchmarking service that combines a rich meta model, low marginal cost and ease of use, which drastically reduces the time and cost to define, adapt and run a benchmark.

I. INTRODUCTION

The benchmarks defined by the TPC consortium [1] and their derivatives [2], [3] provide the de-facto standard in evaluating and comparing the performance of database systems.

Yet, running these benchmarks incurs a significant effort: A range of tools need to be co-ordinated to run the actual workloads, to modify the workloads according to specific distributions, and to visualize the results. A considerable amount of state (e.g., input data) needs to be generated and managed. Judging from our own experience in the database development team at SAP, typically a large number of scripts written in different languages are applied to implement benchmarks.

The problem of defining and running benchmarks has been recognized by both the research community and commercial vendors, and has led to a wide range of tools. Most of these, however, are frameworks that focus solely on the ad-hoc execution of a particular kind of benchmark. Examples of such frameworks include SIMS [4] (local resources), OLTP-Benchmark [5] (OLTP), YCSB [6] (cloud) and PolePosition [7] (OR mapping). Quest Benchmark Factory [8] is a commercial product which supports the definition of benchmarks by means of a rich-client program. Yet, its scripting approach leads to a rather limited reusability and extendibility beyond predefined components. Lately, there have been two approaches to offer benchmarking as a service: XQBench [9], based on XCheck [10] and Liquid Benchmarking [11]. Both approaches, however, aim at non-relational data (XML, RDF) and provide limited meta models and execution flexibility.

In order to consolidate and unify various benchmarks and to simplify the development of new benchmarks, we developed a

generic benchmarking service, which is demonstrated here for the first time. Deploying a long-running service instead of an ad-hoc script collection or program has numerous advantages:

- *Collection* of benchmark artifacts (such as workload generators and queries), benchmark definitions and results
- *Analysis* of collected data for comparison with reference results or performance regression tests
- *Automation* of common tasks such as running experiments, regression analysis or detecting performance bugs
- *Small incremental costs* to define individual benchmark artifacts and to add a new database server or users

Our demo also provides a number of technical contributions:

- A *rich meta model* to express all aspects of benchmarking
- A *complete end-to-end solution* with default implementations for most of the common functionality
- An *extensible architecture* to include custom components
- An *easy-to-use web-based UI*, fully supporting benchmark definition and result analysis

The service can easily be deployed in the cloud. Due to the sensitive nature of benchmarking (limitations on result publishing, license fees, data privacy, etc.), however, we expect it to be used in rather restricted settings and private clouds.

In this demonstration, users will be able to interact with the system, register database servers, define new benchmarks, adjust data generators or SQL statements and understand the effects using various graphical representations.

II. CONCEPTS

The goals we have defined for the benchmarking service require several aspects of conceptual underpinning. First we describe how a benchmark is modeled in our service. Next, we sketch the architecture of the system. We continue with a description of the web-based user interface of the benchmarking service and explain how a benchmark can be executed.

A. Modeling a Benchmark

Since our benchmarking service aims to combine flexibility with rich data operations and user guidance, a comprehensive and expressive model is required. The key benefit of this meta model is that artifacts (i.e., components of a benchmark) can be parameterized, stored and reused. The intuitive definition of these artifacts is achieved by a web-based UI, which also supports archiving and comparing results.

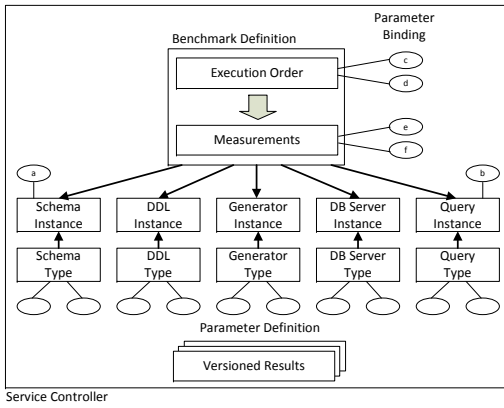


Fig. 1. Data Model

As visualized by the abstract data model (Figure 1), a *benchmark definition* is a combination of several artifact types: *Schema Definition*, *DDL Tuning*, *Data Generators*, *Database Servers*, *Query Set*, *Execution Order*. A simplified example of such a benchmark definition is (“TPC-H schema”, “Index on L_SHIPDATE”, “TPC-H dbgen: Scale 100”, “SAP HANA, PostgreSQL”, “Q1,Q5”, “uniform mix”). In addition to general artifact selection, most of these artifacts can be parameterized.

Generally speaking, a *benchmark* can be seen as a subset of the cross-product of all the artifact types and parameters. Given the possibly large design space, we introduced two means of structuring: 1) *Templates* define the type of a benchmark. Examples of such templates include “a parameterized query on a server (one curve per parameter)” or “several grouped generator runs (one curve per server and query)”. 2) *Measurements* are a grouping of artifacts along particular aspects (yielding, for instance, a line in a graph for a query, scaled over the database size). The known set of artifacts, possible parameters and templates provide information to the GUI to let the user intuitively design and run benchmarks.

The artifacts of a benchmark are described as follows:

Schema Definition. The aim of the schema meta model is to provide abstract information on the data model of individual benchmarks (such as TPC-H), in particular on tables, columns, data types and constraints. This information can be exploited in various ways, among them: 1) Generating DDL statements for creating tables (with metadata specific for a database server type) 2) Generating consistent data-preserving constraints and relationships. In terms of parameterization, we allow the user to choose which columns are being used for each experiment.

DDL Tuning. Beside the schema, there are many aspects in DDLs which can affect performance. We separate them from the schema, so as to provide more flexibility in benchmark design and execution. Typical “tuning” DDL aspects include index creation, materialized views and partitioning. Given the abstract modeling of the schema and the tuning, the system can create both combined and incremental DDL statements at different states within a running experiment.

Data Generator. A data generator can be applied before the execution of an SQL statement in order to populate the database instance with an experimental data set. Different types of data generators are supported and may be combined:

- 1) *Predefined generators* for common benchmarks (e.g., all the TPC benchmarks), supporting the parameters given in the benchmark specification.
- 2) *Generic user-defined generators*: a built-in generator using information from data definition and database server information, covering common aspects such as size, value distribution and correlation between the tables. Furthermore, referential integrity constraints and arbitrary join paths with a chosen selectivity can be defined. All these aspects are exposed as parameters.
- 3) *Custom generators*: Specific requirements can be expressed in the service as custom classes or by calling an external tool (such as [12]). Parameters of these tools need to be specified for the integration into the service.

Database Servers. Since our benchmarking service aims at supporting a multitude of different database servers, the meta model needs to cover three aspects: 1) Capabilities of the database systems involved such as data types, DML expressions, etc. This information can be utilized to tailor DDL and DML statements. 2) Operational information on how to perform operations on the actual server instances using standard call-level interfaces like JDBC, e.g., establishing a connection, executing a query, interpreting the results, all of which will be relevant when running a benchmark. 3) Tunable parameters that are not reachable via normal DDL statements, such as the “merge interval” of SAP HANA or memory/disk settings of Oracle. Besides custom call-level statements, this may involve a collection of scripts at OS-level access.

Query Set. The set of queries to be executed in an experiment can consist of arbitrary DML statements in their textual form. This includes standard SQL statements like queries, insert, update and delete operations, but also stored procedures or scripts in languages such as PL/SQL or T-SQL. Each statement has a possibly empty set of parameters (including type information) for input and output values, allowing for parameterized queries and reusing the output of one query as input for another. Depending on the specification, these parameters may be applied by text replacement or as invocation-time arguments.

Execution Order. Many benchmarks do not consider individual queries in isolation; instead, queries are combined at varying levels of complexity. The meta model of the benchmarking service provides two means to express such interactions: 1) For workloads that consider state changes explicitly, an ordering of the query set may be given. 2) For workloads which combine multiple queries with different cost or characteristics, a query mix can be specified. Once more, a built-in model and driver provide the means to define common aspects like the distribution of query types or their timing.

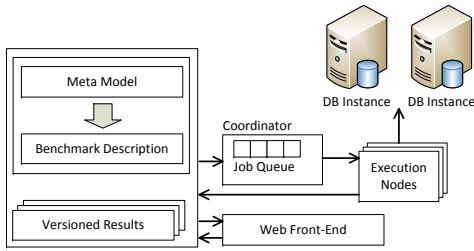


Fig. 2. System Architecture of the Benchmarking Service

Custom query mix drivers may be included to manage those requirements which are not expressible by standard settings.

The entire meta model (artifacts and benchmark specifications) as well as results are stored in a *versioned* database. With this versioning we can track how interactions among artifacts have developed. Furthermore, artifacts can have *variants*, e.g., custom queries for specific DBMSs if automatic tailoring from meta model data is not sufficient.

B. System Architecture

A distributed architecture, as shown in Figure 2, was chosen for the service: A central *Service Controller* keeps track of the meta model instances, which includes both the actual artifacts and the results. The process of running an experiment is controlled by a *Coordinator Node*, which contains a queue of benchmarks that are about to be executed, distributes jobs and detects node failures. The benchmarks are run on several *Execution Nodes* in parallel to simulate a multi-user workload or speed up measurements. Each execution node in turn may distribute the measurements over several database servers. Database servers can be accessed at different levels, mainly using call-level interfaces such as JDBC with queries and statements derived from workloads and DBMS metadata. And, when necessary, at the OS level using scripts to start/stop databases and perform external tuning. Clearly, more access rights provide more precise control of the execution.

The usage model assumes a benchmark cluster or a “private” cloud setting. Using it in a public cloud is possible, but problematic due to the legal and financial constraints of benchmarking (commercial) DBMS: Benchmarking results must not be published without explicit permissions by the vendors. Running DBMS instances in the cloud incurs additional licensing fees, while running DBMSs on customer premises and accessing them from the cloud is often prohibited for security reasons.

C. Web-Based User Interface

The service controller provides a web front-end (Figure 3) for the definition of artifacts and visualization of results. This GUI leverages the powerful meta model introduced in Section II-A by exposing the various kinds of artifact types.

For the definition of a benchmark, the web front-end allows the user to combine artifacts and specify parameters. For instance, in the TPC-H data definition, the configuration of a benchmark is done in multiple steps. It includes the known DDL tuning options, the parameters for generating input data

and the queries with their parameters. Once an experiment has finished, its results can be compared to similar experiments.

The web front-end provides a comprehensive access to features, models and results of the experiments. Yet, the system supports including custom code and classes for special problems such as specific parameter distributions or complex and state-dependent conditional execution orders.

D. Running Benchmarks

Benchmarking is inherently expensive, as it involves computation- and data-intensive tasks like running input data generators and loading the generated data into databases. Furthermore, our definition of a benchmark as a cross-product of its contributing artifacts and their parameters, provides great flexibility, but can possibly entail high cost. The benchmarking service contains several strategies to cope with these costs: Users can specify directly or implicitly (using a template) which execution flow to follow. We apply a number of optimizations: The sequence of steps can be modified to reuse previous, costly stages (like dataset creation or DB loading). In addition, the data generator performs caching and pipelining (depending on the setting) to reduce memory and/or CPU costs. Whenever possible, the controller distributes and parallelizes steps as to take advantage of available nodes. A typical flow of such an execution flow is shown in Figure 4. Correctness of the results and precision of measurements can be ensured. Within an experiment, measurements are performed on a “hot” database and repeated several times to achieve stable results. Users may specify reference results against which the output values of queries are to be compared.

III. DEMONSTRATION

Our demonstration will consist of three parts: 1) introduction, 2) benchmark definition 3) running the experiment.

A few words on the benchmark setup: While the benchmarking service is inherently designed for distribution, we will make our demonstration more tangible by using two local machines – one running the GUI and service controller, the

Create new benchmark	
Benchmark name:	TPC-H
Benchmark type:	grouped bar charts: single result per server and query
Schema:	TPC-H
Metric:	Execution time
Number of repeats:	13
Number of discards:	1
Prepopulate generator:	TPC-H dbgen
Scaling factor:	100.0
Save values:	no
Create tables:	yes
Execution frequency:	once
Timeout (seconds):	1000
Is public:	yes
Is active:	yes
E-Mail notification:	yes
* Required field	
<input type="button" value="Next"/>	

Fig. 3. Screenshot of the Web Front-End

other hosting execution nodes and DB Server installations. If possible, we will also include nodes from our lab.

A. Part 1: Introduction and Motivation

We will begin the demonstration by illustrating some scripts that were deployed at SAP to run standard benchmarks such as TPC-C and TPC-H before our benchmarking service was introduced. By trying to tune some benchmark parameters in the script, we will show how inconvenient and error-prone defining a benchmark was in the past. Based on a standard benchmark, we will demonstrate how easily the configuration can be adapted using the new system.

B. Part 2: Defining a Benchmark and Adding DBMSs

In the second part of the demonstration we will show how the web-interface can be used to create a new benchmark from scratch in only a few minutes and run on different DBMSs. The example use-case will be the evaluation of join queries.

First, the audience will see how several different database servers can be registered with the system. We will continue by creating a new database schema related to the synthetic data used to micro-benchmark the join queries. This schema, in this instance, will include two tables with attributes of different types and a foreign-key constraint. In the following step we will create a user defined data generator for this schema together with the audience. We will agree on different types of distributions for each field of the tables (e.g., uniform distribution, Zipf distribution and sequences) to assess how the joins are to be processed on skewed data. The generator will populate the database with values fulfilling the constraints and distribution. The demonstration will continue with the creation of SQL join queries with parameters influencing aspects such as the selectivity of the join predicate. Next, servers, a schema, a generator and queries will be combined and a new benchmark will be defined. We will add a measurement to the benchmark for each configuration of the join queries, each DBMS, and we will choose a suitable visualization.

C. Part 3: Running the Experiment and Results

In the final part of the demonstration the benchmark will be executed. The progress of the running experiment will be monitored using the web-interface. When the experiment is finished, an e-mail with the link to the result page will be sent. Next, together with the audience, we will examine the plot and interpret the result. We will find out that it is necessary to adapt the data type and the selectivity of the join attributes, and we will rerun the experiment. In addition, one of the database servers will need to be tuned by adding an index. The effect of this modification can be visualized by comparing the new result with the previous one in a single diagram.

A most relevant aspect of the demonstration is that we will create a new benchmark from scratch, taking into account the feedback and suggestions from the audience. It will become evident that these measurements can be defined within the benchmarking service in only a few minutes as opposed to several hours for the manual implementation of the benchmark

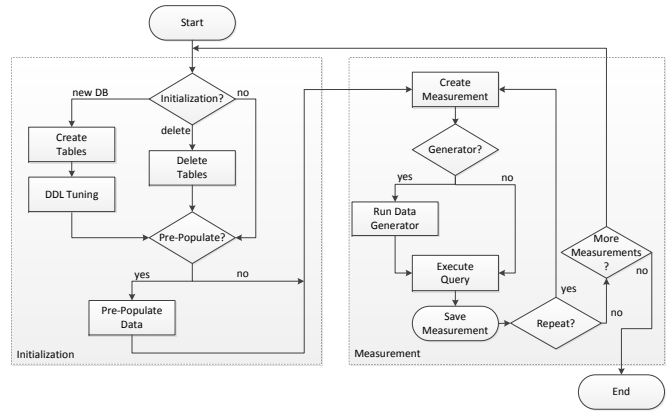


Fig. 4. Steps for Running an Experiment

in a traditional scripting language. All reoccurring tasks, such as plot generation, storing, archiving and comparing results can be configured and are handled by the application automatically.

IV. CONCLUSION

This demonstration presents a generic benchmarking service and its benefits over previous benchmarking frameworks: An expressive meta model supports defining and reusing artifacts and benchmark definitions by capturing the relevant properties and “powering” a user-friendly, yet effective web GUI. Furthermore, the service incurs only a small cost when adding new resources and provides automation of common tasks. Rich visualization and analysis capabilities complete the picture.

A previous version of this service is used extensively by the database engineering group at SAP. We noticed developers now run significantly more experiments than before since benchmarks are much easier to define, leading to the discovery and correction of regressions and correctness bugs. The tool has also been applied to non-relational workloads, in particular XML and XQuery. In terms of scalability, we have so far reached a scaling factor of 100 for TPC-H. We plan to release an open-source version [13] of the benchmarking service soon.

REFERENCES

- [1] “TPC transaction processing performance council,” Website, 2012, <http://www.tpc.org/>.
- [2] F. Funke, A. Kemper, and T. Neumann, “Benchmarking Hybrid OLTP&OLAP Database Systems,” in *BTW*, 2011.
- [3] P. O’Neil, E. O’Neil, and X. Chen, “The star schema benchmark,” University of Massachusetts, Boston, Tech. Rep., 2007.
- [4] H. J. Jeong and S. H. Lee, “An integrated benchmark suite for database systems,” in *ISDB*, 2002, pp. 74–79.
- [5] “OLTP-Benchmark,” Website, 2012, <http://www.oltpbenchmark.com/>.
- [6] B. F. Cooper et al., “Benchmarking cloud serving systems with YCSB,” in *SoCC*, 2010.
- [7] “PolePosition,” Website, 2012, <http://www.polepos.org/>.
- [8] “Quest Benchmark Factory,” Website, 2012, <http://www.quest.com/benchmark-factory/>.
- [9] P. M. Fischer, “XQBench - A XQuery Benchmarking Service,” in *XML Prague*, 2011, pp. 341–355.
- [10] L. Afanasiev et al., “XCheck: A Platform for Benchmarking XQuery Engines,” in *VLDB*, 2006.
- [11] S. Sakr and F. Casati, “Liquid Benchmarks: Benchmarking-as-a-Service,” in *JCDL*, 2011.
- [12] J. Gray et al., “Quickly generating billion-record synthetic databases,” in *SIGMOD Conference*, 1994.
- [13] “A Generic Database Benchmarking Service,” Project Website, 2013, <http://www.benchmarking-service.org/>.