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 or TPC-H provide a way of comparing the performance of different systems. In addition, micro benchmarks can be used 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. In this demo, we introduce a generic benchmarking service that combines a rich meta model, low marginal cost and ease of use to reduce drastically the time and cost to define, adapt and run a benchmark.

I. Introduction

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

Yet, running these standard benchmarks incurs a significant effort since usually a range of tools need to be co-ordinated to run the actual workloads, modify the workloads parameters according to specific distributions and visualize the results. Judging from our own experience at the database development team at SAP, typically a large number of scripts written in different programming languages are applied to implement multiple benchmarks.

The problem of defining and running benchmarks has been recognized by both the research community and commercial vendors, leading 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] (benchmarks with local resources), OLTP-Benchmark [5] (OLTP benchmarks), YCSB [6] (benchmarks in the 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, the scripting approach of Benchmark Factory leads to a limited reusability and extendibility of its 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 for 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 like running experiments, regression analysis or detection of performance bugs
- *Small incremental costs* to define and tune individual benchmark artifacts, to add a new database server or to brief and add a new user to the system

Besides these conceptual aspects, our demo provides a number of interesting 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 benchmark components
- An *easy-to-use web-based UI*, fully supporting benchmark definition and result analysis

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

II. Concepts

The goals we have set 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 comparison of results.

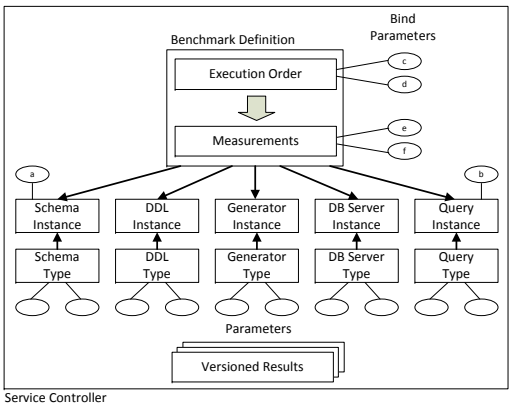

Fig. 1. Data Model

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 (e.g., the date range for a parameter in a TPC-H query).

Generally speaking, a *benchmark* can be seen as a subset of the cross-product of all the artifacts types and parameters. Given the possibly large design space, we have introduced two means of structuring: 1) *Templates* at varying levels of abstraction 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 a particular result set such as 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 in the following:

Data Definition. The aim of the *Data Definition* 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 multiple ways, among them: 1) Generating DDL statements for creating tables (with meta-data specific for each individual database server type) 2) Generating consistent data preserving constraints and relationships. In terms of parameterization, we allow to choose which of the columns are actually being used for each experiment.

DDL Tuning. Besides the basic data definition, there are many aspects in DDLs which can affect performance. We conceptually separate them from the data definition, 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 basic data definitions and the tuning, the system can create both combined and incremental DDL statement 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) A generic user-defined generator: a built-in generator using information from data definition and database server information, covering common aspects such as the 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 this tools need to be specified for the integration into the service.

Database Servers. Since our benchmarking service aims to support 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, column 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, e.g., establishing a connection, executing a query, interpreting the results, all of which will be relevant when running a benchmark 3) Tunables that are not reachable via normal DDL statements, such as the "merge interval" of SAP HANA or memory/disk settings of Oracle.

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 query specifics, these parameters may be applied by text replacement or as invocation-time arguments.

Execution Order. Many benchmarks do not consider individual queries in isolation, but 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, a ordering of the query set may be given 2) For

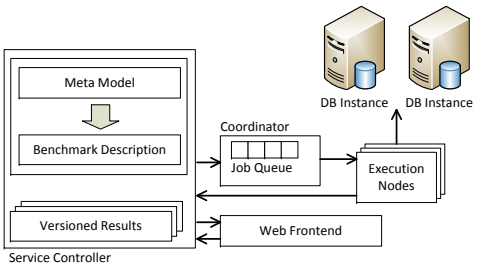

Fig. 2. System Architecture of the Benchmarking Service

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. 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 the results are stored in a *versioned* database. With this versioning it is possible to keep track how artifacts and results have evolved and at which version certain interactions have happened. Furthermore, artifacts can have *variants*, e.g., custom queries for specific database servers 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 benchmarking 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, detects node failures and timeouts. The benchmarks are run on several *execution nodes* in parallel in order to simulate a multi-user workload or speed up measurements. Each execution node in turn may distribute the actual database measurements over several database servers. Users can register their database servers with different levels of access: as a normal user via JDBC/ODBC, as a database administrative user or as a OS user. The more access they give to the service, the more precisely the execution flow can be controlled.

The usage model for the system architecture is a benchmark cluster in each department of a company. In the future, we plan to deploy a benchmark as a service in the cloud.

C. Web-Based User Interface

The service controller provides a web frontend (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 frontend allows the user to combine artifacts and specify the parameters. In the example of the TPC-H data definition, the configuration of a benchmark is completed in multiple steps and includes the known DDL tuning options, the parameters for generating benchmarking data and the queries with their parameters. Once an experiment has finished, its results can be compared to similar experiments.

The web frontend 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

The definition of a benchmark as a cross-product of its contributing artifacts and their parameters provides a great amount of flexibility, but might also incur significant cost if executed naively. The benchmarking service contains several strategies to cope with this possible cost: Users can specify directly or implicitly (using a template) which execution flow to follow. The service itself applies 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. Even more important than performance are correctness of results and precision of resource measurements. The service takes various means to ensure those: 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 being 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 other hosting execution nodes and DB Server installations. If possible, we will also include nodes from our lab.

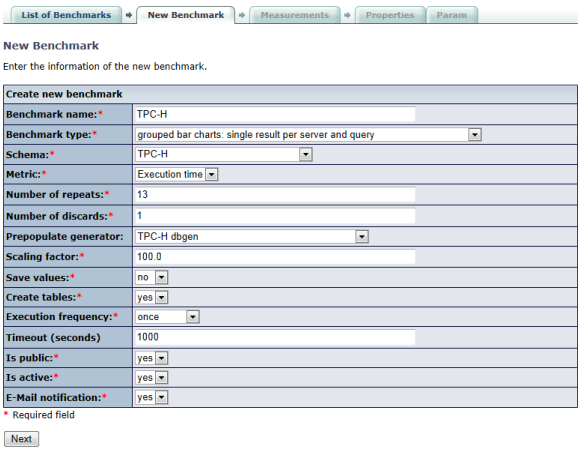

Fig. 3. Screenshot of the Web Frontend

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 the 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

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. One use-case will be the evaluation of join queries in different database systems.

First, the audience will see how new 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, e.g., 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 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 and 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, we will examine the plot with the audience 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 rerun the experiment. In addition, one of the database servers has 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.

An important aspect of the demonstration is that we will create a new benchmark from scratch taking into account the feedback and suggestions from the audience. The result will be that these measurements can be defined within the benchmarking service in a few minutes as opposed to several hours for the manual implementation of the benchmark 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.

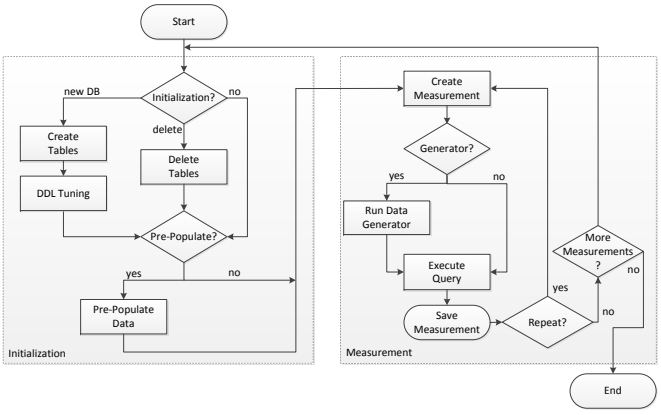

Fig. 4. Steps for Running an Experiment

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 in 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 with 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 scalability, we have so far reached a scaling factor of 100 for TPC-H. We plan to release an open-source version 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, pp. 390–409.

[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, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010, pp. 143–154.

[7] "PolePosition - open source database benchmark," 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, M. Franceschet, M. Marx, and E. Zimuel, "XCheck: A Platform for Benchmarking XQuery Engines," in *VLDB*, 2006, pp. 1247–1250.

[11] S. Sakr and F. Casati, "Liquid Benchmarks: Benchmarking-as-a-Service," in *JCDL*, 2011.

[12] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *SIGMOD Conference*, 1994, pp. 243–252.