MulTe: A Multi-Tenancy Database Benchmark Framework—Implementation Details

— work in progress —

Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner

Dresden University of Technology
Database Technology Group
Dresden, Germany
{tim.kiefer, benjamin.schlegel, wolfgang.lehner}@tu-dresden.de

Abstract. MulTe is a benchmark framework for multi-tenancy database management systems. The framework allows to define and generate multiple database workloads based on legacy database benchmarks. It helps to load them as tenants into a database system. Finally, a workload driver that is part of the framework executes the tenants' queries and analyzes the results. MulTe hence helps to quickly create and run new benchmarks for multi-tenancy databases. MulTe was first introduced in 2012 at the TPC Technology Conference[1]. A main intention of providing MulTe publicly was to encourage the community to use it for their research and also to possibly extend it according to their needs. This documentation, which is supplemental to the original paper, shall help to understand design decisions and implementation details of MulTe. It is based on version 1.0 which is available at the time of the paper publication (August, 2012).

[1] Kiefer, T., Schlegel, B., Lehner, W.: MulTe: A Multi-Tenancy Database Benchmark Frameworkt. In: Proceedings of the 4th TPC Technology Conference on Performance Evaluation & Benchmarking - TPCTC '12, Istanbul, Turkey (2012)

Table of Contents

1 Introduction . . . 2
2 Quick Start: Step-by-step Example . . . 4
2.1 Pre-requisites . . . 4
2.2 Installing and Running MulTe . . . 4
2.3 Using a Remote MySQL Server . . . 5
2.4 Known Issues . . . 5
3 Benchmark Framework Overview . . . 6
3.1 System Requirements . . . 6
3.2 Detailed benchmark workflow . . . 7
4 Python Framework Implementation . . . 10
4.1 Framework Structure . . . 10
4.2 Framework Configuration . . . 11
4.3 Setup Builders . . . 11
4.4 Tenant Generators . . . 15
4.5 Database Executors . . . 17
5 Java Workload Driver Implementation . . . 18
5.1 Comparison with the Original TPoX Workload Driver . . . 18
5.2 Workload Driver Configuration . . . 20
5.3 High-Level Class Structure . . . 24
5.4 Statistics Output Format and Location . . . 25
5.5 Information and Error Logging with Log4j . . . 27
5.6 Access to the Database: DatabaseOperations . . . 27
6 Analysis and Charting . . . 29
7 Custom Framework Extensions . . . 29
7.1 Adding a New Workload . . . 30
7.2 Adding a New Benchmark . . . 31
7.3 Adding a New Database System . . . 31
8 Future Work . . . 32

1 Introduction

Academia and industry have shown increasing interest in multi-tenancy in relational databases for the last couple of years. Ever increasing capabilities and capacities of modern hardware easily allow for multiple database applications with moderate requirements to share one system. At the same time, cloud computing leads to outsourcing of many applications to service architectures (IaaS, SaaS), which in turn leads to offerings for relational databases hosted in the cloud as well (e.g., Microsoft Windows/SQL Azure or Amazon RDS). We refer to any database system that accommodates multiple tenants by means of virtualization and resource sharing, either on a single machine or on a hosted infrastructure of machines, as a multi-tenancy database management system

(MT-DBMS). Building MT-DBMSs leads to interesting challenges like, (1) assigning logical resources to physical ones; (2) configuring physical systems (e.g., database design, tuning parameters); and (3) balancing load across physical resources, all of which require thorough testing to ensure scalability and system quality.

In the original paper[2], we propose and provide methodology, workflow, and associated tools to benchmark MT-DBMSs. There, we also introduce and provide the framework MulTe[3] which allows for generating multi-tenancy benchmarks quickly and easily. This supplemental documentation intents to detail many of the design and implementation decisions that led to MulTe but were beyond the scope of the paper. We would like to help the reader to understand the existing implementation and thereby help to modify and extend the framework according to the specific requirements at hand. We assume the reader to be familiar with the general concepts and ideas of MulTe that are introduced in the above mentioned paper.

As shown in Figure 1, our approach in MulTe is to re-use existing benchmarks, including their schemas, data, and queries/statements and to generate instances of these benchmarks to represent different tenants. Each tenant is given an individual, time-dependent workload that reflects a user's behavior. A workload driver is used to run all tenants' workloads against any multi-tenancy database system and to collect all execution statistics and benchmark metrics.

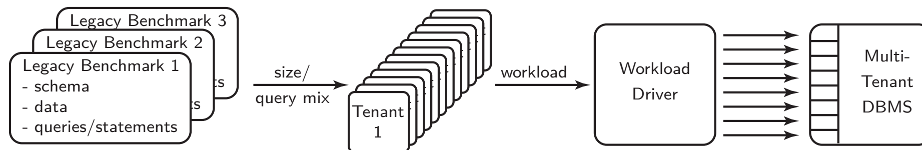

Fig. 1: Multi-Tenancy Benchmark Workflow (from the original paper)

With the framework's help, it remains the user's responsibility to define meaningful tenants (including their dynamic workloads) for the system and performance questions at hand. The user also needs to think about the performance metrics (and aggregations thereof) that help to answer said questions.

In the remainder of this documentation, we first provide a step-by-step example in Section 2. The example is intended as a quick start and can be read independently of the rest of the document. Afterwards, we start with a framework overview in Section 3 before we dive into the implementation details in Sections 4–6. We dedicate Section 7 to what we think are likely extension of MulTe, before we list our planned future improvements in Section 8.

[2] Kiefer, T., Schlegel, B., Lehner, W.: MulTe: A Multi-Tenancy Database Benchmark Frameworkt. In: Proceedings of the 4th TPC Technology Conference on Performance Evaluation & Benchmarking - TPCTC '12, Istanbul, Turkey (2012)

[3] http://wwwdb.inf.tu-dresden.de/research-projects/projects/multe/

2 Quick Start: Step-by-step Example

In this Section, we present a step-by-step example that shows how to prepare the system and how to install MulTe. The process of creating and running a very simple multi-tenancy benchmark with MulTe is outlined. The example can be used as a first hands-on exercise with the framework, but more advanced features and extensions are beyond the scope of the example.

Some of the instructions are specific to the Ubuntu operating system that we use, the reader may replace these steps with the ones appropriate for the system at hand. For reference, we use a standard installation of Ubuntu 11.10 Desktop in a virtual machine to test the step-by-step example.

2.1 Pre-requisites

The system that runs MulTe must provide the following software packages. You can install them following these steps. You can skip any step if the according software is already installed.

1. Install Python (versions 2.7.2 and up should work, earlier versions might work)
2. Install MySQL server (e.g., sudo apt-get install mysql-server). Optionally, create a user other than `root` with appropriate rights to create and load databases.
3. Install Oracle Java JDK (versions 6 and up should work), e.g.,
   - Download and unpack tar.gz archive to `/home/java_jdk`
   - Add `/home/java_jdk/bin` to `PATH` variable
   - Make sure that `/home/java_jdk/bin` is the first path in the `PATH` variable to prevent version mismatches between the java compiler and the runtime environment installed with Ubuntu (double-check if you get a `java.lang.UnsupportedClassVersionError`)
   - Set the `JAVA_HOME` system variable to `/home/java_jdk`
4. Install `make`, `gcc`, and `ant`
5. Download and compile TPC-H datagen, i.e.,
   - Download and extract the archive from the TPC website
   - Rename `makefile.suite` to `makefile`
   - Modify `makefile` (set `CC` to “gcc”, `DATABASE` to, e.g., “SQLSERVER”, `MACHINE` to “LINUX”, and `WORKLOAD` to “TPCH”)
   - Run `make`

2.2 Installing and Running MulTe

The following steps are required to install and run the MulTe framework:

1. Download and unpack the MulTe framework archive
2. Change directory to `multe/multe_workload_driver/build`
3. Run `ant` to compile the Java code and create a JAR-archive

4. Change directory to `multe/setup_example`
5. Modify `run.py`—set paths and parameters (e.g., username/password for MySQL). Make sure that all placeholders `[...]` are replaced with actual values.
6. Copy the MulTe Python framework (`multe/multe_python_framework`) to a location, where Python searches for modules. Alternatively, add a symbolic link to the framework (if not already present):
   `ln -s ../multe_python_framework multe_python_framework`
7. Execute `run.py` with the following parameters:
   `./run.py --prepare --run --analyze}`
   See "Known Issues" in Section 2.3, if you encounter the following problem:
   `ERROR 29 (HY000): File '...' not found (Errcode: 13)`
   Two databases (`tenant_0000` and `tenant_0001`) are populated, the workload driver is executed for one minute and the results are printed on screen.
8. (optional) Configure the workload suite description, which was automatically generated in `[path_to_tenants]/suite.xml`, e.g., to set the duration of the test-run (default, 1 minute) or to configure a *baseline run*. Re-run `run.py` without the first parameter `--prepare`

2.3 Using a Remote MySQL Server

The bulk-load facility of MySQL (`LOAD DATA INFILE...`) assumes the files to load to be accessible by the server (and not by the client executing the command). This needs to be considered when the MulTe framework is executed on a different machine than the one that hosts the MySQL server. One solution is to set up a shared network directory that both machines can access. Another solution is to have MySQL transfer the data via the connection between client and server by calling `LOAD DATA LOCAL INFILE...`. Then, the files to load can be located on the client's machine. This second solution is prepared in
`multe_pyton_framwork/executors/MySQLDatabaseExecutor_remote.py`.
To use it, modify the import section of `run.py` accordingly.

```
Replace
   from multe_python_framework.executors.MySQLDatabaseExecutor
   import MySQLDatabaseExecutor

with
   from multe_python_framework.executors.MySQLDatabaseExecutor_remote
   import MySQLDatabaseExecutor
```

2.4 Known Issues

The combination of MySQL and Ubuntu can cause a known "File not found" error when the `run.py` script tries to bulk-load data. The error is caused by

the AppArmor security module that prevents MySQL from reading from the specified directory[4].

There are two known workarounds for this problem:

1) Follow the instructions in the forum thread to re-configure AppArmor such that MySQL can read from the directory

```
sudo vi /etc/apparmor.d/usr.sbin.mysql

/usr/sbin/mysqld {
...
/var/log/mysql/ r,
/var/log/mysql/* rw,
/var/run/mysqld/mysqld.pid w,
/var/run/mysqld/mysqld.sock w,
/[multe_base_directory]/** rw
}

sudo vi/etc/init.d/apparmor reload
```

2) A workaround is to use `LOAD DATA LOCAL INFILE...` to bulk-load data as if the MySQL server was on a remote machine. To use this solution follow the instructions in Section 2.3 (*Using a Remote MySQL Server*).

3 Benchmark Framework Overview

Based on the idea to re-use existing database benchmarks, the general workflow of benchmarks built with MulTe is shown in Figure 2. The characteristics of a tenant are defined with a basic set of parameters. This includes, e.g., the benchmark type (like TPC-H), the size of the raw data, and the query mix. Based on these descriptions, instances of the benchmarks are generated and loaded as tenants into the MT-DBMS. All three steps are supported by a set of Python scripts. Once populated, a Java workload driver runs the tenants' workloads against the MT-DBMS and collects all relevant performance statistics. These statistics are analyzed, evaluated, and (in future) visualized with Python.

3.1 System Requirements

We use Python and Java for the implementation of MulTe for their availability and (relative) platform independence. All our experiments were conducted with Python 2.7.2 and Java 6, but earlier versions of Python should work as well.[5]

[4] cf, http://ubuntuforums.org/showthread.php?t=822084

[5] The Java `ServiceLoader` facility used to implement and provide DBMS specific database operations was introduced in Java version 6. Hence, earlier versions of Java are not supported.

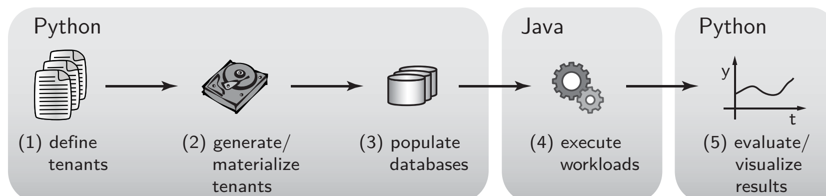

Fig. 2: MulTe—General Benchmark Workflow

Beyond these major components, the benchmark system must provide the build tool `ant` to build the workload driver.

Additional requirements may depend on the database system and legacy database benchmark that shall be used. E.g., to use TPC-H, the system must provide `gcc` and `make` to compile the TPC-H data generator. The database system for a benchmark may be hosted on the same or a remote machine. Hence, the system may or may not have to provide the database system software. But in both cases, the database system's **client application** must be available on the machine that runs MulTe, e.g., the command line clients of MySQL or PostgreSQL.

MulTe provides exemplary implementations for tenant (instance) generators and database executers. So far, MulTe supports the TPC-H benchmark on MySQL or PostgreSQL databases. This leads to the additional requirements that the TPC-H data generator (dbgen) as well as the MySQL (or PostgreSQL) command line tool need to be available in the system.

3.2 Detailed benchmark workflow

A detailed workflow of MulTe, including intermediate formats and references to the various sections that provide details about the components, is shown in Figure 3.

The first step of any benchmark is to define tenants, i.e., their workload types and characteristics. This definition is done by so called *setup builders*, which are described in Section 4.3. A setup builder's task is to come up with a set of parameters for each tenant. To define arbitrarily realistic tenants, setup builders may use the full expressiveness of the Python language. The result of any setup builder is a list of tenants (i.e., tenant descriptions) in the form of `Tenant` classes (defined in `multe_python_framework/__init__.py`).

The generated list of tenant descriptions is handed over to the second component in the framework, a *tenant generator* (described in Section 4.4). A tenant generator's task is to materialize tenants based on the given descriptions. The materialization consists of the tenant's raw data, schema information, queries, and workload descriptions. Subsequent framework components assume certain directories to be part of each tenant's materialization. All directories that a tenant must provide (in red) as well as example files from the TPC-H benchmark

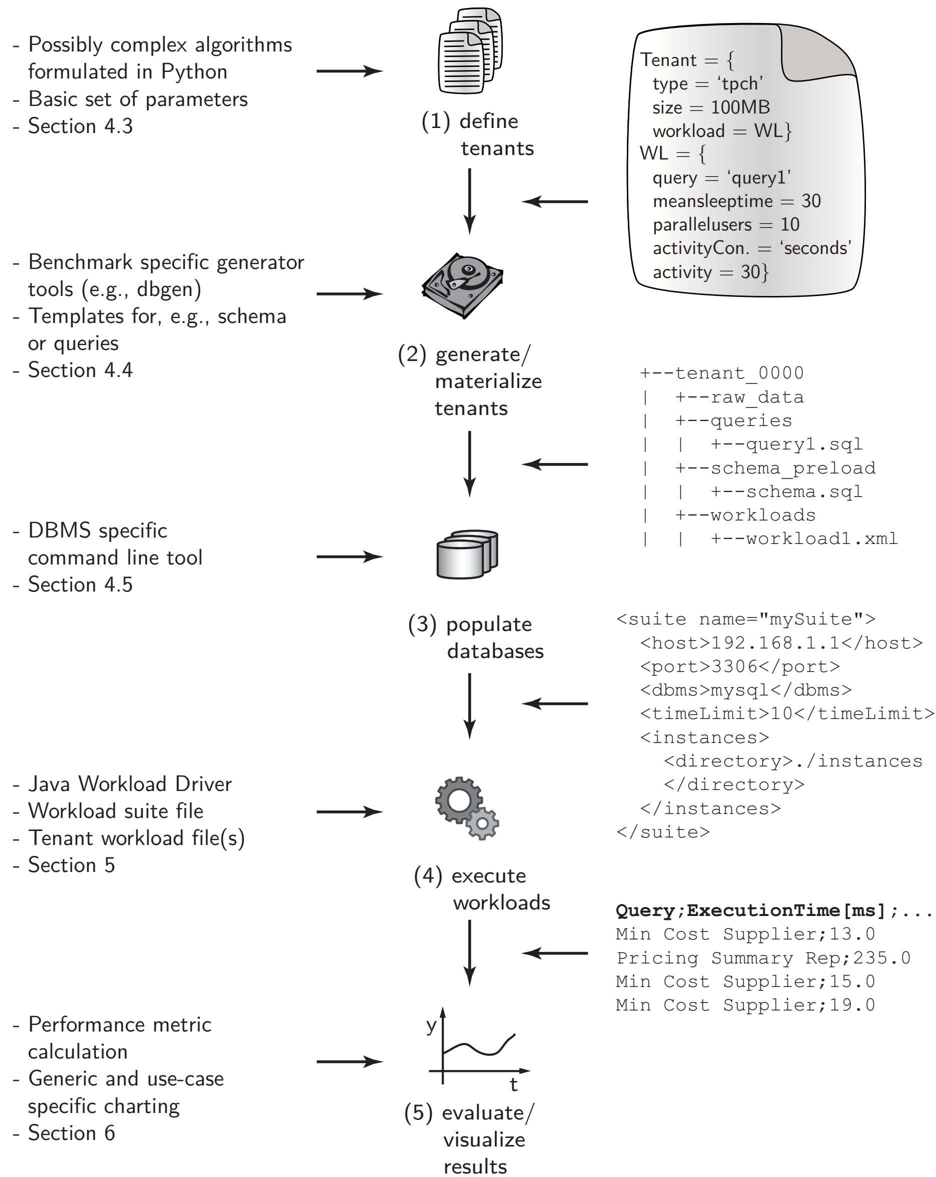

Fig. 3: MulTe—Detailed Benchmark Workflow

are shown in Figure 4. MulTe uses benchmark-supplied tools and user-provided templates to generate, e.g., data and schema information.

```
tenant_0000/
├── data/
│   ├── customer.tbl
│   ├── lineitem.tbl
│   ├── nation.tbl
│   ├── orders.tbl
│   ├── partsupp.tbl
│   ├── part.tbl
│   ├── region.tbl
│   └── supplier.tbl
├── queries/
│   ├── 01.sql
│   ├── 02.sql
│   ├── 03.sql
│   └── ...
├── schema_postload/
│   └── 02_tpch_constraints.sql
├── schema_preload/
│   └── 01_tpch_schema.sql
├── workloads/
│   ├── pvalues/
│   │   ├── q02-regions.txt
│   │   ├── q02-syllable3.txt
│   │   ├── q03-segments.txt
│   │   └── ...
│   └── workload_01.xml
└── info.txt
```

Fig. 4: MulTe Tenant Directory Structure

Taking the materialized tenants, i.e., the directories, as input, the third step is to populate the databases. This step is performed by so called *database executors* which are described in Section 4.5. Current implementations of database executors use the database systems' command line clients to connect to the database and to execute statements in the database. The results of this third step are fully populated tenant databases that await to be queried. Additionally, the Python framework uses the provided information about the tenants to create a workload suite configuration file that is input for the workload driver.

The execution of a workload is the forth step in any benchmark. MulTe's Java workload driver, described in Section 5, is responsible for the workload execution. The workload driver is generic as it can execute many different workloads. The actual workload to execute for the given benchmark is determined by a global suite configuration file and per tenant workload configuration files. The output of the workload driver is a number of CSV-files that contain query execution times of all queries.

The last step in a benchmark is to analyze, aggregate, and possibly visualize the results. These post-processing actions are performed by so called *result analyzers*. Depending on the benchmark to execute and the performance questions to answer, generic or use-case specific analyzers may be executed. A simple example result analyzer that takes the logged execution times an aggregates them in a table is described in Section 6.

4 Python Framework Implementation

The Python component of MulTe consists of two parts, the `multe_python_framework` and an example script `run.py`. The Python framework contains re-usable modules (classes and functions) to define, generate, and load tenants as well as to analyze workload driver execution results (described in Section 6. At this time, MulTe can be used to generate instances of the TPC-H benchmark and to load them to MySQL or PostgreSQL databases. Additionally, two exemplary functions are provided that show how to define tenants and one exemplary function is provided to do a basic analysis of the results. The `run.py` script demonstrates how the Python framework can be configured and used. We will describe the framework in more detail in the following sections.

4.1 Framework Structure

The high-level structure of the part of the MulTe Python framework that is responsible for setting up a benchmark is shown in Figure 5[6]. The corresponding directory and file structure of the framework is shown in Figure 6. The framework provides a single entry point, the function `run`. This method prepares the system for the framework execution and then calls (in this particular order) a *setup builder*, a *tenant generator*, and a *database executor*. All three components are described in detail in the following sections. The components can be replaced with different implementations that extend the framework to, e.g., support another MT-DBMS. The usage of the `run` function is shown in Listing 1. The listing also shows what steps are executed inside the `run` function. It can be useful to call these steps manually to leave things out in the process of experimenting with new components.

[6] Functions to analyze benchmark results are, although also written in Python and part of the framework, separately described in Section 6. They require executions of the workload driver first and can be invoked anytime and possibly multiple times afterwards, independent of the rest of the framework.

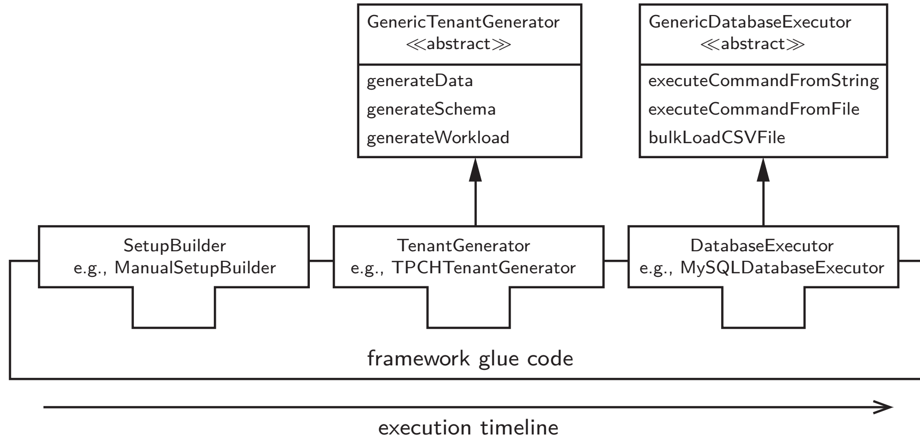

Fig. 5: Python Scripts Overview

4.2 Framework Configuration

The MulTe Python framework uses a standard Python dictionary variable named `config` to provide global parameters to all components (cf. Listing 1). The framework user must provide this configuration—it is the first parameter for all framework functions. An example configuration comes with the framework and is also printed in Listing 2. Values in brackets must be replaced with values that reflect the actual system.

The flexibility of the dictionary variable allows future extensions of the framework to simply add global configuration parameters. Existing components will ignore parameters that they do not expect or need. Framework developers should make it a habit to accept the configuration variable as the first parameter in all functions/methods that can be called from another module.

As a special case, the configuration must contain a key value pair `("quiet": BOOLEAN)`. The value of this parameter will be used throughout the framework to decide whether to ask the user certain questions (if `False`). It can be useful to suppress any questions and to rely on default decisions (if `quiet` equals `True`), e.g., to run the script with `nohup`. The exemplary `run.py` accepts the command line flag `-quiet` to enforce quiet mode by setting the respective parameter.

4.3 Setup Builders

Setup Builders are components in MulTe that define characteristics for the tenants, i.e., properties like the benchmark type and size as well as the workload. Given the configuration and any number of additional parameters, a setup builder must return a list of tenants. The Tenant-class definition (in `multe_python_framework/__init__.py`) dictates the properties of each tenant, i.e., the variables that a setup builder must set. Tenants have the following properties:

```
multe_python_framework/
├── databaseexecutors/
│   ├── __init__.py
│   ├── MySQLDatabaseExecutor.py
│   ├── MySQLDatabaseExecutor_remote.py
│   └── PostgresDatabaseExecutor.py
├── resultanalyzers/
│   ├── __init__.py
│   └── ExampleResultAnalyzers.py
├── setupbuilders/
│   ├── __init__.py
│   └── ExampleSetupBuilders.py
├── tenantgenerators/
│   ├── __init__.py
│   └── TPCHTenantGenerator.py
├── __init__.py
├── distributions
└── license
```

Fig. 6: MuLTe Python framework, directory structure

Listing 1: Framework usage example

```python
import multe_python_framework
from multe_python_framework.setupbuilders.ExampleSetupBuilders
  import manualTenants
from multe_python_framework.databaseexecutors.MySQLDatabaseExecutor
  import MySQLDatabaseExecutor

multe_python_framework.run(config, manualTenants, MySQLDatabaseExecutor)

# is equivalent to

tenants = manualTenants(config)
multe_python_framework.prepare(config)
multe_python_framework.generate(config, tenants)
multe_python_framework.load(config, tenants, MySQLDatabaseExecutor)
multe_python_framework.createWorkloadSuite(config)
```

Listing 2: Framework configuration example

```
# configure the following variables to meet your local environment
# [...] are placeholders for actual values and need to be replaced

BASE_PATH = '[BASE_PATH]'   # framework base path

config = {
  # system variables
  "host": '[HOST]',
  "port": '[PORT]',
  "dbms": '[DBMS]', # e.g., 'mysql', or 'postgres'

  # general variables for dbms
  # comment out, if no user/pw is needed for the dbms
  "dbmsUser": '[USER]',
  "dbmsPassword": '[PASSWORD]',
  # custom flags for the dbms
  # "dbmsCustomParameters": ['-x', 'value'],

  # environment variables for mysql
  "mysqlExecutable": 'mysql',

  # environment variables for postgres
  "postgresExecutable": 'psql',

  # framework variables
  "rawDataPath": os.path.join(BASE_PATH, 'raw_data_repository'),
  "tenantPath": os.path.join(BASE_PATH, 'tenants'),
  "templatePath": os.path.join(BASE_PATH, 'templates'),

  # environment variables for tpch
  "dbgenPath": '[DBGEN_PATH]',
  "dbgenExecutable": os.path.join(['DBGEN_PATH'], 'dbgen'),

  # workload driver variables
  "workloadSuiteTemplate": 'workload_suite_template.xml',

  # special parameter to flag "quiet" runs without user interaction
  "quiet": False,
}
```

- `name`: A tenant's unique name. The name is used to identify the tenant and to determine the directory where the tenant's data is materialized on disk (i.e., `config['tenantPath']/tenantName`). The tenant's name is also used to name the database that is created for the tenant.
- `generator`: An instance of a *tenant generator*, e.g., the `TPCHTenantGenerator` (cf. Section 4.4). The tenant generator defines the type of the benchmark.
- `size`: The tenant's size in megabytes. The size is determined by the size of the raw data (on disk) that is loaded into the database
- `workload`: A tenant's workload (see below).

A tenant's workload is also set by the setup builder. The workload is a dictionary variable and hence flexible with respect to the description. In the first version of MulTe, a workload must contain the following parameters:

- `query`: The name of an XML-file that contains the query mix to execute. The XML-file has to follow the conventions of the Java workload driver.
- `meansleeptime`: The time in seconds that a tenant is idle between bursts of activity.
- `parallelusers`: The number of active users that query the database in parallel during activity bursts.
- `activeconstraint`: Either `seconds` or `transactions`—defines the limiting counter of an activity burst.
- `active`: Together with `activeconstraint`, this variable defines the length of an activity burst, either as the number of seconds to run or the number of transactions to execute.

The parameters that define a workload may change in future releases. However, any changes to the workload description must be incorporated in the Java workload driver as well to make sure that the correct workload is executed.

Setup builders in MulTe do not have any state or lifetime, hence they are simple functions (as opposed to classes). Any function that returns a list of tenants can be used as a setup builder. MulTe provides two simple exemplary setup builders to demonstrate how tenants can be defined. The first setup builder `manualTenants` defines two small tenants with fixed parameters. The second setup builder `randomIndependentTenants` accepts the number of tenants to define as a parameter. It then defines that many tenants with characteristics drawn from independent random variables. The tenants that are built with `randomIndependentTenants` are not intended to be meaningful with respect to any multi-tenancy benchmark. The example's intention is rather to assist the user in the process of defining own setup builders.

Please note: If a setup builder requires any parameters besides the configuration variable, these parameters must be fixed before the framework's `run` function is called. The Python `functool` module can be used to fix parameters as shown in Listing 3

Listing 3: Example of a setup builder that requires parameters

```
import functools
import multe_python_framework
import multe_python_framework.setupbuilders.ExampleSetupBuilders

setupBuilder = multe_python_framework.setupbuilders...
  ExampleSetupBuilders.randomIndependentTenants
setupBuilder = functools.partial(setupBuilder, count = 10)
multe_python_framework.run(config, setupBuilder, dbExecutor)
```

4.4 Tenant Generators

Tenant Generators are components in MulTe that take a tenant's description and generate (i.e., materialize on disk) all corresponding data for the tenant. The data to generate includes the benchmark's raw data to load into the database, the schema information related to the benchmark data, optional indexes, and queries as well as workload files that are used as input for the workload driver. Different methods can fulfill a tenant generator's tasks, i.e., providing a tenant's data. The exemplary tenant generator in MulTe—the `TPCHTenantGenerator`—uses both, a data generating tool of the TPC-H benchmark and templates, to materialize a tenant on disk.

The tenant generator is part of the tenant's description. In the process of generating tenants, the framework calls each tenant's generator to materialize this particular tenant. This implies that different tenants can have different tenant generators and hence be based on different legacy benchmarks.

To use the provided `TPCHTenantGenerator`, simply create an instance of the corresponding class as demonstrated in both of the example setup builders. The tenant generator's class constructor is used to provide the framework configuration and the tenant description.

To extend the framework with a new tenant generator, the user must provide a class that implements the interface of `GenericTenantGenerator` (i.e., in Python terms: extend this class and override all methods). The generated data must be structured as it was shown earlier, in Figure 4. The following methods must be implemented (with one exception):

- `generateInfoFile(self)`: This method creates a file named `info.txt` in the tenant's base directory. The file contains any information about the tenant that the user may find useful, e.g. the tenant's benchmark type and size. The information files of all tenants can be used to generate an overview of the benchmark setup. So far, no other component of MulTe depends on the existence of these information files, hence overriding this method is optional.
- `generateData(self)`: This method generates the tenant's raw data depending on the legacy benchmark to implement. Most benchmarks provide dedicated data generation tools or fixed datasets for this purpose. The

TPC-H benchmark, e.g., provides `dbgen` to generate datasets. The task of the `generateData` method is to invoke such tools correctly and to move the generated data to the tenant's directory. To save disk space and reduce generation time, the `TPCHTenantGenerator` uses a data repository and keeps once generated data of certain sizes for later re-use (see best practices section below).

- `generateSchema(self, schema_name):` The `generateSchema` method creates the schema that describes the data in the database. More generally, we subsume all statements under *schema* that must be executed before the data can be loaded or after the data has been loaded. Before the data can be loaded, tables need to be defined. Additionally, value-range or referential constraints may be created. After the data was loaded into the database, the user may want to create indexes or, e.g., run a command that gathers statistics on the data. The TPC-H tenant generator uses templates to provide the schema information for all tenants, i.e., schema files that contain all table definitions are provided once and only copied for each tenant. If necessary, a schema name can be used to personalize the template for each tenant (see best practices section below)
- `generateWorkload(self):` The `generateWorkload` method of a tenant generator is responsible for creating all necessary workload information in the tenant's directory. The workload information consists of two parts. First, the query/queries that together comprise the query mix. Second, all online information that is required by the workload driver while executing the workload. This includes, e.g., value ranges for parameter markers or the number of parallel users that execute queries. The TPC-H tenant generator again uses templates to generate the workload corresponding to the tenant's description. Queries for the benchmark can be used without modification and simply copied to the tenant's directory. The workload is picked from a set of XML-files that is provided. Again, tenant-specific parameters are replaced with actual values when the template is copied to the tenant's directory.

Best practice (raw data repository): To generate raw data for a large number of tenants can be a time-consuming job. In addition, the raw data for many tenants may require a considerable amount of disk space which adds to the space that is needed for the actual databases. To ease both difficulties, the TPCH Tenant Generator uses a *raw data repository* to store TPC-H datasets of different sizes. Whenever a tenant is generated with a certain size, the data repository is checked first. If the dataset of that size is found in the repository, only a symbolic link is created in the tenant's directory. If the dataset cannot be found in the repository, it is generated using the `dbgen` tool and moved to the raw data repository. The actual tenant, again, only has a symbolic link to the data. To use such a repository has been useful in practice, especially with repeated generation of certain tenant sizes and with large numbers of tenants that have the same size. Additionally, the data repository can further be optimized by, e.g., placing it on a dedicated, fast disk (cf. framework configuration variable).

Best practice (templates): Many parts of a tenant's materialization on disk are either the same for all tenants or differ only slightly. For example, the schema information to create the tables for the TPC-H benchmark are practically the same for all tenants. Also the set of queries that can be executed is fixed and does not change for different tenants. For all such parts, it has been useful to use templates that can either be copied to the tenant directory directly or with slight modifications. To replace pre-known placeholders in schema or workload files, the string template facilities in Python are used (refer to the `TPCHTenantGenerator` source code for an example). Placeholder in the template files start with a `$`-sign.

MulTe provides a template directory with templates for all supported benchmark/database combinations, i.e., TPC-H on MySQL and PostgreSQL. It is necessary to have templates for all combinations of benchmarks and databases that the user wants to use. Obviously, each benchmark type needs own templates for schema definitions, queries that can be executed, workloads that shall run and so on. Unfortunately, it is not sufficient to have a template per benchmark type, because each database management system has a slightly different syntax for the same task or query. Take, e.g., MySQL and PostgreSQL as a reference. Both DBMS differ in the syntax for date functions that are used in the TPC-H queries. Consequently, if a user wants to follow this best practice when adding a new tenant generator, all templates must be provided besides the tenant generator class.

4.5 Database Executors

Database Executors are components in MulTe that encapsulate all database accesses. All steps that are necessary to load a tenant's data to the database, e.g., to create tables, to create indexes, and to load the actual data, are performed by a database executor. The database executor is an interchangeable component that can be replaced to support another DBMS. So far, MulTe provides two database executors, one for MySQL and the other one for PostgreSQL.

Similar to tenant generators, MulTe provides a simple interface for all database executors. Any newly added executor should extend the `GenericDatabaseExecutor` and implement the following three methods:

- `executeCommandFromString(self, command, **args)`: This method takes a SQL command as a parameter and executes it in the database system. Additionally, any number of arguments can be provided that shall be used for the execution of the command (see best practices section below).
- `executeCommandFromFile(self, file_name, **args)`: Similar to the first method, `executeCommandFromFile` executes a number of commands in the database system. The difference is, that commands are read from a file that may contain any number of SQL statements that shall be executed. Again, any number of additional arguments may be provided.
- `bulkLoadCSVFile(self, file_name, db, table='')`: The last method of the database executor interface (`bulkLoadCSVFile`) is used to load a benchmark's raw data into the database. The method's parameters specify, which

file contains the data and into which database/table they shall be loaded. Both database executors in MulTe use the bulk load facilities of the corresponding database systems to load data (as opposed to inserting records one by one). To load the data at once usually has a performance benefit over loading records individually. However, a database executor may fall back to inserting all records individually if the database system does not support the bulk load of data.

Best practice (database command line tools): There are different possibilities to access a database and execute commands. One way would be to use the Python Database API to directly connect to the database. However, we decided to use the database systems' command line tools instead.[7] Both database systems that are currently supported, provide such command line tools to connect to a database: MySQL has a tool named `mysql`, PostgreSQL provides a similar tool `psql`. Having such tools, the task of the MulTe database executor is to first assemble the necessary command line options to execute a given command in the correct database on the correct server with the correct parameters. Second, the database executor calls the command line tool with the assembled options. The above mentioned additional parameters that can be provided as parameters to the database executor are simply added to the call of the command line tool.

5 Java Workload Driver Implementation

The workload driver in MulTe is a Java tool that takes a workload description, connects to the MT-DBMS, and executes queries according to the workload description. During workload execution, the driver collects statistics about the query execution times that can be used to analyze the system's performance. The workload driver, although being one tool, simulates multiple tenants that execute queries independently. The workload description consists of two parts, a global description for the whole workload suite and individual descriptions for each tenant's workload. All files that describe the workload are generated by MulTe and require little to no manual modification.

The workload driver is a complex piece of software and we do not have the resources to document it in all detail. Instead, we will outline some of the main properties and features in the following sections and refer the reader to the source code for any further studies of the workload driver.

5.1 Comparison with the Original TPoX Workload Driver

The workload driver is a heavily modified and extended version of the workload driver that was developed for the TPoX XML database benchmark[8]—specifically, version 2.1 of TPoX. A good documentation of the workload driver

[7] This decision was made with the intention to keep the Python scripts simple. The user may find it useful to use the Python Database API instead. We do not see any reason why this may be a better or worse thing to do.

[8] `http://tpox.sourceforge.net/`

with many examples can be found on the TPoX website. Many of the descriptions and properties hold for the MulTe workload driver as well.[9]

The original idea was to be able to flexibly describe a workload and to have a driver that takes the description and executes queries accordingly while collecting runtime statistics at the same time. The same requirements hold for the workload driver that is used in MulTe. As mentioned, many of the documented features of the TPoX workload driver can still be found in the MulTe workload driver. For example, the workload description and execution are almost the same, hence the same rich variety of workloads can be executed with MulTe. The most important difference between the TPoX and the MulTe workload driver is the following:

> **The MulTe workload driver executes multiple individual workloads of various tenants at the same time. Basically, multiple instances of the workload driver are executed internally, all of them with the abilities of the original driver. Each tenant can connect to an own database and run an individual workload regardless of the other tenants.**

Additionally, there are some minor differences with respect to implementation details

- The MulTe workload driver supports a new parameter type `DIFF` that can be used like `SAME`. The new type means that a parameter is taken from the same range of parameters as the one it is referring two, but has a different actual value than the first parameter.
- The MulTe workload driver supports *tuples* as parameter values, e.g., taken from files. Together with the `same` parameter type, one parameter value can be the first part of the tuple while another actual parameter value can be the second part (cf. the workload templates for TPC-H for an example).
- The MulTe workload driver's execution parameters are configured with one additional global XML configuration file. This file replaces many of the command line parameters of the original workload driver (cf. Section 5.2).
- The ability to collect and report statistics is significantly reduced in the MulTe workload driver. There is no online statistics output to the screen anymore. We do not see a useful way yet to output statistics of many tenants at the same time. Instead, execution times are collected internally and written to files. Any evaluation or post-processing of these statistics is delegated to other components of MulTe (cf. Sections 5.4 and 6).
- The MulTe workload driver uses log4j to control all of the logging and user communication. Hence, the user can flexibly decide to direct the output to file or screen (cf. Section 5.5).

[9] There is no strong connection between the development teams of TPoX and MulTe. We simply like the tool and decided to use/extend it for our needs. Any shortcomings/bugs of the MulTe workload driver do not reflect on the TPoX workload driver, we may have introduced them in the refactoring process.

- The implementation of the database operations has changed. It is now easier to support a new database system by providing a separate JAR-file (cf. Section 5.6)

5.2 Workload Driver Configuration

The MulTe workload driver is a versatile tool that is supposed to drive all different kinds of workloads and benchmarks. The exact specifications of these workloads are beyond our knowledge. Hence, to allow for a maximum of flexibility there are various options to configure the behavior of the workload driver.

There are three levels of configuration options in the MulTe workload driver: (1) command line parameters, (2) the workload suite configuration file, and (3) individual workload configuration files.

Command line parameters: It is a design desicion to keep the command line parameters at an absolute minimum—only one parameter is mandatory. Only parameters that immediately influence the workload driver execution are implemented as command line parameters. Right now there are the following parameters:

`-suite <suite>`	This only mandatory parameter configures the name of the workload suite configuration file, hence the workload suite to execute.
`-o <output>`	This optional parameter sets the root directory for all output that is written during the workload execution, i.e., all execution statistics. If this parameter is not set, output is written to a directory named `output` under the current directory.
`-noValidation`	This flag disables the validation of the configuration XML-files.
`-h`	Prints an overview of all parameters.

Workload suite configuration file: The configuration file for the whole workload suite (containing many tenant workloads) contains all global parameters like the server to connect to or the duration of a benchmark run. The suite configuration file is generated by MulTe from a template and located in the root directory of all materialized tenants (default: `[multe]/tenants/suite.xml`). This file can be modified if needed to execute a different workload suite, e.g., to include or exclude certain tenants. An example of a suite configuration file is shown in Listing 4. All possible elements of the suite configuration file are shown in the listing, many of them are self-explaining.

All other parameters have the following meaning and can be used as follows:

- `databaseSystem`: The `databaseSystem` value must be a string that any of the available `DatabaseOperations` implementations accepts (cf. Section 5.6).

Listing 4: Example workload suite configuration file

```
<?xml version="1.0" encoding="UTF-8"?>

<suite name="exampleWorkloadSuite"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <host>192.168.1.1</host>
  <port>3306</port>
  <userId>user1</userId>
  <password>user1</password>
  <databaseSystem>mysql</databaseSystem>
  <comment>example workload suite</comment>
  <executionTimeLimit unit="min">1</executionTimeLimit>
  <!-- baselineRun>1</baselineRun -->

  <tenants>
      <!-- paths must be absolute or relative
           to this description file! -->

    <directory subdirpattern="tenant_*/workloads"
               filepattern="workload*.xml">
      /home/multeuser/multe/tenants
    </directory>

    <!-- alternatively, for selected tenants only -->
    <!-- <file>tenant_0000/workloads/workload_1.xml</file> -->
    <!-- <file>tenant_0001/workloads/workload_1.xml</file> -->
  </tenants>

</suite>
```

- `executionTimeLimit:` The execution time for the whole suite can be limited in two units: `(m|min|minute)` and `(h|hrs|hour)`. Note that the workload driver is not able to abort long-running queries at the end of the workload execution. Hence, the execution time limit cannot always be enforced.
- `baselineRun:` The `baselineRun` value can be set to collect baseline execution times, i.e., execution times in a simulated single-tenant environment. With this value set, all tenants execute their workloads sequentially and each query is executed as many times as the value of this parameter.
- `tenants:` The `tenants` element specifies the location of the tenants (i.e., their workload description files) that shall be executed. Tenants can either be specified as a directory with subdirectories to search (allowing wildcards) or as individual workload file locations. Combinations of both methods are possible as well. In Listing 4, the workload driver will search tenants' workload files of the format `workload*.xml` in the directories that match `/home/multeuser/multe/tenants/tenant_*/workloads/`.

The workload suite configuration file is validated against a schema definition[10]. The schema file is integrated in the JAR-file when the workload driver is built. Hence, you have to rebuild the workload driver (at least the JAR-file) after you have made changes to the schema.

Workload configuration files The workload configuration file specifies an individual tenant's workload. The workload consists of one or many SQL statements (including INSERT, DELETE, UPDATE statements and procedure calls) and the activity description. The workload driver picks the transaction to execute from the list—based on assigned weights—whenever the tenant is active. The activity description dictates, when the particular tenant is active and when the tenant is idle. Listing 5 shows an example workload configuration. The first four elements of the workload description detail the tenant and his activity (these elements are new in the MulTe workload driver and were not part of TPoX). The `transactions` element is used to specify the query mix for the tenant. We refer to the TPoX workload driver documentation[11] for very comprehensive and detailed examples on how transactions can be defined to execute a rich variety of queries. The reader may also have a look at the workload configuration files that are provided with MulTe as templates for TPC-H.

The workload configuration file, like the suite configuration file, is validated against a schema definition[12]. The schema file is integrated in the JAR-file when the workload driver is built. Hence, you have to rebuild the workload driver (at least the JAR-file) after you have made changes to the schema.

[10] `[multe]/multe_workload_driver/config/suite_properties.xsd`
[11] `http://tpox.svn.sourceforge.net/viewvc/tpox/TPoX21/documentation/WorkloadDriverUsage_v2.1.pdf`, Section 5
[12] `[multe]/multe_workload_driver/config/workload_properties.xsd`

Listing 5: Example workload configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<workload name="tpch query 1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="workload_properties.xsd">

  <database>tenant_0000</database>

  <parallelusers>2</parallelusers>
  <when_active constraint="seconds">20</when_active>

  <meansleeptime>10</meansleeptime>

  <transactions>

    <!-- #### transaction 01 ##########################-->
    <transaction weight="1" name="Pricing Summary Report">
      <location>
        /home/multeuser/multe/tenants/tenant_0000/queries/01.sql
      </location>
      <parameters>
        <parameter>
          <uniformint min="60" max="120"/>
        </parameter>
      </parameters>
    </transaction>

  </transactions>
</workload>
```

5.3 High-Level Class Structure

It is beyond the scope of this documentation to explain all the components and many classes of the workload driver in detail. The interested reader may have a look at the source code and the generated javadoc documentation. Here, we only give a high-level overview of the packages and classes that form the MulTe workload driver.

The packages of the workload driver are shown in Table 1. The implementation of database operations is detailed in Section 5.6. The relationships of the core classes are shown in Figure 7. The `WorkloadSuite` contains the `main` method and is therefore the entry point. It initializes and uses `GlobalSuiteParameters` for the benchmarks, i.e. all parameters taken from the workload suite configuration file (here `suite.xml`). The `WorkloadSuite` furthermore drives multiple tenants (`WorkloadInstance`) which are implemented as Java threads. Each `WorkloadInstance` in turn uses multiple `ConcurrentUsers` (also threads) to simulate a tenant's parallel connections. The `ConcurrentUser` class is responsible for picking the next transaction of a tenant's query mix and for delegating the actual execution of the statement to the `DatabaseOperations` class. A tenant's workload parameters—organized in `WorkloadInstanceParameters`—are taken from the workload configuration XML-file that is part of the tenant definition (here `workload??.xml`). The `WorkloadInstanceAdministration` manages the `WorkloadInstanceParameters` for all tenants.

Package `org.tud.multe.*`	Description
`databaseoperations`	The abstract class for all operations that directly interact with the database as well as all implementations of the class (cf. Section 5.6).
`workload.core`	The main classes of the workload driver, e.g., classes for the workload suite and workload instances (cf. Figure 7).
`workload.parameter`	Classes that implement the different parameter types, e.g., `FILE` or `UNIFORMINT`, that can be used in transactions.
`workload.statistics`	Classes to collect and output execution statistics. Many of these classes are deprecated and may be removed in future releases.
`workload.transaction`	Classes that implement an abstract transaction, i.e., an arbitrary SQL statement.
`workload.util`	Utility classes to, e.g., count the number of started/terminated worker threads.

Table 1: Workload driver package overview

Activity implementation: Activities, i.e., time-dependent workloads, play an important role in MulTe. It is possible to define activities for individual

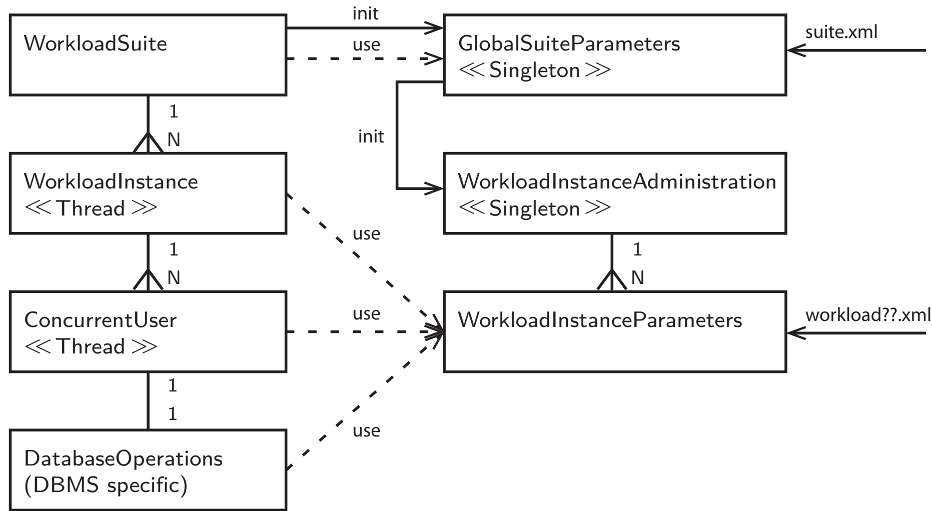

Fig. 7: Java Workload Driver Overview

tenants with the Python framework. To actually implement these activities is a task of the workload driver. In the first version of MulTe, a tenant's activity consists of alternating active and idle periods. During an active period, multiple users can query the database in parallel. The implementation of such activities is located in two classes: `WorkloadInstance` and `ParallelUser`. Each `WorkloadInstance`, representing a tenant, sleeps for a certain amount of time to simulate an idle period. Afterwards, the `WorkloadInstance` spawns as many `ConcurrentUser` threads as there are parallel users during the tenant's active period. Each `ConcurrentUser` is initialized with the activity constraint, i.e., either a number of transactions to execute or an amount of time to run for, and is responsible for executing the planned activity. Afterwards, the `ConcurrentUser` terminates itself and an idle period follows. Any changes to the way that activities are defined and described (in the Python framework) must be implemented here as well.

5.4 Statistics Output Format and Location

The MulTe workload driver collects execution statistics and writes them to the file system. Because the analysis and interpretation of results significantly depends on the kind of benchmark that is executed as well as this benchmarks intention, we decided to simplify the output as much as possible. The MulTe workload driver collects query execution times and writes them as they are to files, i.e., in raw format. Any analysis like aggregations of the results are deferred to dedicated analysis components, which we decided to implement in Python.

An example output directory is shown in Figure 8. The output is by default written to a directory named `output` under the current directory. This direc-

tory is created if it does not already exist. The root directory can be changed with the command line option `-o` of the workload driver. Under the root directory, a dedicated directory (containing the workload execution time and date) is created for each workload driver execution and hence benchmark run. An exception are baseline runs; their output is written to a directory named `baseline` under the root directory. In the directory for the specific test run, a file named `global_stats.csv` as well as subdirectories for each tenant are created. The subdirectories in turn contain one file each, named `stats.csv`. Both CSV-files contain the execution time for one query in each line.

```
output/
└─ baseline/
│   ├─ global_stats.csv
│   ├─ tenant_0000/
│   │   └─ stats.csv
│   └─ tenant_0001/
│       └─ stats.csv
└─ output2012_07_18_151517/
    ├─ global_stats.csv
    ├─ tenant_0000/
    │   └─ stats.csv
    └─ tenant_0001/
        └─ stats.csv
```

Fig. 8: Example output directory structure

An example `stats.csv` file is shown in Listing 6 (line breaks are added for readability). Each line contains the execution time of one query with the following additional information (in this particular order):

- The executing tenant
- The transaction name as it is defined in the workload description file
- The start time of the transaction
- The end time of the transaction
- The duration of the transaction execution in milliseconds

The workload driver uses a blocking queue to write all statistics to files. All concurrent users that belong to one tenant add elements (i.e., lines) to the queue and the stats writer constantly takes elements from the queue and writes them to the file. The composition of each line in the statistics file that contains the above mentioned information takes place in the `processTransaction` method in the `ConcurrentUser` class. Any changes to the output format, e.g., to add additional statistics about the query execution, should be placed there.

Listing 6: Example statistics output file

```
tenantName;transactionName;startTime;endTime;duration
tenant_0000;National Market Share;2012-08-23 14:28:59.149;...
  2012-08-23 14:28:59.782;633.0
tenant_0000;National Market Share;2012-08-23 14:28:59.150;...
  2012-08-23 14:28:59.798;648.0
tenant_0001;National Market Share;2012-08-23 14:28:59.149;...
  2012-08-23 14:28:59.933;784.0
tenant_0001;National Market Share;2012-08-23 14:28:59.150;...
  2012-08-23 14:28:59.936;786.0
```

5.5 Information and Error Logging with Log4j

The MulTe workload driver uses Apache log4j[13] to communicate everything from debug information to error messages to the user. The advantage of log4j is that it can easily be configured to send the output to the console or a log file or other output writers. It is also possible to log events of different components at different detail levels. We refer to the log4j website for a detailed documentation of the features and configuration options of the logging facility. Here, we only explain, where the configuration file is located and where the workload driver searches for it.

The directory `[multe]/multe_workload_driver/config` contains a file named `log4j.porperties`. This file is used for the default configuration and also included in the JAR-file during the built process. Upon execution, the workload driver will first look for a `log4j.properties` file in the current directory and use this file if it exists. Only if it does not exist, the workload driver will use the configuration inside the JAR-file. The workload driver does never directly use the configuration in the `config` directory. Hence, if you change the configuration there, you have to rebuild the JAR-file to make the changes work.

As default, the MulTe workload driver logs all messages of levels `INFO` and up directly to the console.

5.6 Access to the Database: DatabaseOperations

A principal design goal of the MulTe framework is that new benchmarks and database systems can easily be supported. The MulTe workload driver is very flexible and powerful in defining all kinds of different workloads, hence new benchmarks can usually be added without any changes to the workload driver itself. Adding a new database system requires modifications/extensions to the workload driver code. Thanks to the JDBC database interface, the necessary steps to add a new database system to the workload driver are minimal.

The abstract class `GenericDatabaseOperations` implements a *template method* pattern. Almost all operations related to the database, like `establishConnection`,

[13] `http://logging.apache.org/log4j/1.2/`

Listing 7: MySQL database operations implementation

```java
public class MySQLDatabaseOperations extends GenericDatabaseOperations {

  protected void loadJDBCDriver() throws InstantiationException,
      IllegalAccessException, ClassNotFoundException {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
  }

  protected String getConnectionURL() {
    if (this.host == null) {
      return "jdbc:mysql:" + databaseName;
    } else {
      return "jdbc:mysql://" + host + ":" + port + "/" + databaseName;
    }
  }

  protected GenericDatabaseOperations getDatabaseOperations(String name) {
    if (name.toUpperCase().equals("MYSQL")) {
      return new MySQLDatabaseOperations();
    } else {
      return null;
    }
  }
}
```

`prepareStatements`, and `executeTransaction`, are implemented and can be used (or overridden if necessary) by descendants of the class. Only the following three abstract methods need to be provided by any DBMS specific extension of `GenericDatabaseOperations`:

- `loadJDBCDriver()`: This method loads the appropriate JDBC driver for the given database system. The JDBC driver needs to be in the classpath of the workload driver.
- `getConnectionURL()`: This method composes and returns the connection string/URL that is used to connect to the database system.
- `getDatabaseOperations(String name)`: This method is part of the ServiceLoader facility (see below) that is used to find and use implementations of `GenericDatabaseOperations`. It checks the requested database system (`name`) and returns an instance of itself if it fits or `null` otherwise.

An example for an extension of `GenericDatabaseOperations`, i.e., for the MySQL database system, is shown in Listing 7.

To allow users to add new database systems without modifying the existing code, the workload driver uses a service loader facility. The generic database operations are seen as a *service* and the DBMS specific implementations as *service*

providers. Thereby, the java ServiceLoader[14] can be used to locate and load new service providers, i.e., extensions for new database systems. We refer to the ServiceLoader documentation for details on its functionality. In a nutshell, a new class like the one shown in Listing 7 that extends GenericDatabaseOperations and hence implements the three abstract methods must be written and archived in a JAR-file. The JAR-file must also contain a file named GenericDatabaseOperations under META-INF/services that has the name of the newly written class as the only content. This file allows the service loader to locate and load the service provider. The JAR-file must be available in the workload driver's classpath. To use the newly added database system, the workload suite configuration must refer to it by its unique name, the same (user-defined) string that is checked for in the getDatabaseOperations method.

An example of how to use ant to compile and archive an own extension of GenericDatabaseOperations can be found in the build.xml of the workload driver. There, all provided database operations extensions, i.e., for MySQL, PostgreSQL, DB2, MS SQL Server, and Oracle, are processed to generate the necessary JAR-files.

6 Analysis and Charting

Analyzing, printing, and charting of benchmark results is also part of the MulTe framework. Depending on the configuration of the benchmark and the system parameters that shall be tested, quite different forms of analysis and output may be required. The different dimensions include the performance metrics to measure and calculate, the aggregates to build from the metrics, and the charts or visualizations to generate from measured or aggregated metrics. Because MulTe can be used to build all different kinds of benchmarks, the idea is to provide a catalog of analysis components that can be used depending on the actual benchmark. All such components are called result analyzers and kept in the directory resultanalyzers of the MulTe Python framework. So far, MulTe provided a simple table output as an example result analyzer.

Simple Table Output The *simple table output* is an example result analyzer that takes the measured query execution times from the global_stats.csv and computes minimum, maximum, and average query execution times as well as query throughput per tenant and query. An example output of this analyzer is shown in Listing 8.

7 Custom Framework Extensions

In this section, we would like to outline the necessary steps that a developer must make to extend the MulTe framework. We concentrate on those extension that we think are most likely demanded by framework users. The intention of this

[14] http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html

Listing 8: Example table output

```
Pretty-printing results in file:
/home/multeuser/multe/setup_example/output/output2012_08_25_115754/global_stats.csv

Explanation:
  minqet -- minimum query execution time
  maxqet -- maximum query execution time
  avgqet -- average query execution time
  eet    -- experiment execution time

 =========================================================================================================
|   tenant    ||         query          || executions || minqet || maxqet || avgqet || eet  || throughput |
|             ||                        ||            || in ms  || in ms  || in ms  || in s ||  query/s   |
-----------------------------------------------------------------------------------------------------------
| tenant_0000 || Pricing Summary Report ||    315     || 242.0  || 292.0  || 251.6  || 50.7 ||    6.21    |
| tenant_0001 || Minimum Cost Supplier  ||    270     ||  12.0  ||  37.0  ||  21.6  || 43.4 ||    6.22    |
 =========================================================================================================
|   global    ||           *            ||    585     ||  12.0  || 292.0  || 145.4  || 50.7 ||   11.54    |
 =========================================================================================================
```

section is to highlight the components of the framework that need to be touched to implement the extensions. However, details on the specific components are located elsewhere in this documentation or in the source code itself.

7.1 Adding a New Workload

One possible extension to the MulTe framework is to add new workloads based on an existing benchmark. In this case, the benchmark's data and schema already exist and can be re-used. The following changes are necessary to add a new workload:

- **Templates:** The templates, as part of the Python framework, for the specific benchmark and database system in question need to be extended by the new workload. These extensions consist of two parts: the actual SQL statements in the `queries` directory and the workload descriptions in the `workloads` directory.
- **Python:** A new or existing setup builder must be provided (or modified, respectively) that defines tenants that use the new workload(s). The `query` parameter of a tenant's workload must point to any of the new workload description files in the templates directory.
- **Java:** There are usually no modifications of the workload driver necessary to execute a new workload. The new workloads are described in the workload description files and executed by the workload driver as such. An exception of this default case is when a new workload requires a new parameter type for the new query (i.e., when none of the provided parameter types like `uniformint` or `file` fulfill the requirements). In the current version of the workload driver, it is not easily possible to add new parameter types (deeper knowledge of the workload driver code is required... this will hopefully change in future releases). However, since the code is open source, any developer is free to modify the workload driver to any extend ;).

7.2 Adding a New Benchmark

Another likely extension of MulTe is to add a completely new benchmark. The complexity of such an extension greatly depends on the complexity of the new benchmark, e.g., its schema and queries. Also, the availability of data sets or data generators influence the complexity of the framework extension. The following steps are necessary to add a new benchmark:

- **Templates:** New template directories for the new benchmark and all database systems in question need to be created. Each template directory must contain the benchmark's schema information as well as queries and workload descriptions.
- **Python:** A new tenant generator must be provided to generate and materialize instances of the new benchmark. The new tenant generator should extend the generic tenant generator and hence implement all methods to generate data, schema, and workload.
- **Python:** A new or existing setup builder must be provided (or modified, respectively) that defines tenants that use the new benchmark, i.e., the new tenant generator.
- **Java:** There are usually no modifications of the workload driver necessary to execute a new benchmark's workload. We have discussed the exception of this case in the section on adding a new workload.

7.3 Adding a New Database System

A common extension of MulTe will be to support a new database system. The following modifications and extensions need to be done to add a new database system:

- **Templates:** New template directories for the new database system and all benchmarks in question need to be created. Each template directory must contain the benchmark's schema information as well as queries and workload descriptions. Depending on how much the new system's syntax differs from an existing supported database system, it may be possible to simply copy another template directory.
- **Python:** A new database executor must be provided for the new database system. The new executor should extend the generic database executor and implement all methods that are defined there. The implementation of the actual database operations, i.e., either with a command line tool of the database system or with the Python database API, depends on the new system and user's preferences.
- **Java:** The new database system must provide a JDBC interface and driver. The MulTe workload driver must be extended with a new database operations class that loads and uses the new system's JDBC driver. The newly added class must be an extension of `GenericDatabaseOperations`. The new database operations class can be added as a JAR-archive without recompiling the actual workload driver. It therefore must register as a service provider for `GenericDatabaseOperations`.

- **Configuration:** The workload suite configuration must be modified to use the new database system. This happens automatically when the MulTe Python framework is executed and the generated suite configuration is used.

8 Future Work

We actively use MulTe for our own research. We hence gain experience with the framework and constantly improve the code. Additionally, we have multiple ideas on how to further extend the framework in the future.

Benchmark and Database System Extensions: We initially support TPC-H, MySQL, and PostgreSQL partly to make own experiments, partly to test the framework and to evaluate its extensibility. We will certainly add new benchmarks and database systems in the future and also provide these extensions publicly whenever they may be interesting for the community.

Scalability and Parallelization: We have tested MulTe with moderate numbers of tenants (up to 100) and tenant sizes between 10MB and 1GB. Depending on the hardware that is used, generating and loading many tenants is a time-consuming process. Also, driving many tenants' workloads with possibly multiple concurrent users requires the system to spawn and handle a large number of threads. Consequently, the system that executes the benchmark may become the bottleneck long before the (possibly distributed) MT-DBMS reaches its limits.

To overcome such problems and to make sure that MulTe scales with the number of tenants at least as good as the MT-DBMS does, we think about possibilities to parallelize MulTe. Any parallelization strategy must consider the Python part of the framework as well as the workload driver. Since tenants are generally independent from one another and hence *data parallel*, it should be possible to parallelize all steps of the process as long as there are single controlling instances of the Python framework and the workload driver.

Workload Description: A very fundamental design decision in MulTe was with respect to the workload description. The approach to describe tenants individually and to allow bursty workloads with idle periods and active periods alternating seems appropriate for our current research activities. However, other possibilities to describe and execute tenants may become more appealing in the future. We may extend or change the framework to support other workload descriptions, but such changes require various modifications of different parts of the framework.