Bexhoma

Release 0.6

Patrick K. Erdelt

Dec 01, 2022

TABLE OF CONTENTS:

CHAPTER

ONE

BENCHMARK EXPERIMENT HOST MANAGER FOR ORCHESTRATION OF DBMS BENCHMARKING EXPERIMENTS IN CLOUDS

{include} ../README.md

1.1 Example: TPC-H

This example shows how to benchmark 22 reading queries Q1-Q22 derived from TPC-H in MonetDB and PostgreSQL.

> The query file is derived from the TPC-H and as such is not comparable to published TPC-H results, as the query file results do not comply with the TPC-H Specification.

Official TPC-H benchmark - http://www.tpc.org/tpch

1.1.1 Prerequisites

For basic execution of benchmarking we need

- a Kubernetes (K8s) cluster
  - a namespace `my_namespace` in the cluster
  - `kubectl` usable, i.e. access token stored in a default vault like `~/.kube`
  - a persistent volume named `vol-benchmarking` containing the raw TPC-H data in `/data/tpch/SF1/`
- JDBC driver `./monetdb-jdbc-2.29.jar` and `./postgresql-42.2.5.jar`

We need configuration files containing the following informations in a predefined format, c.f. demo file. The demo also includes the necessary settings for some DBMS: MariaDB, MonetDB, MySQL, OmniSci and PostgreSQL.

We may adjust the configuration to match the actual environment. This in particular holds for `imagePullSecrets`, `tolerations` and `nodeSelector` in the YAML files.

For also enabling monitoring we need

- a monitoring instance Prometheus / Grafana that scrapes metrics from `localhost:9300`
- an access token and URL for asking Grafana for metrics https://grafana.com/docs/grafana/latest/http_api/auth/#create-api-token

1.1.2 Perform Benchmark

For performing the experiment we can run the tpch file.

The actual configurations to benchmark are added by

```
config = configurations.default(experiment=experiment, docker='MonetDB', configuration=
↪'MonetDB-{}'.format(cluster_name), alias='DBMS 1')
config = configurations.default(experiment=experiment, docker='PostgreSQL',␣
↪configuration='PostgreSQL-{}'.format(cluster_name), alias='DBMS 2')
```

1.1.2.1 Adjust Parameter

You maybe want to adjust some of the parameters that are set in the file: `python tpch.py -h`

```
usage: tpch.py [-h] [-db] [-c CONNECTION] [-cx CONTEXT] [-e EXPERIMENT] [-d] [-m] [-ms␣
↪MAX_SUT] [-dt]
               [-md MONITORING_DELAY] [-nr NUM_RUN] [-nc NUM_CONFIG] [-ne NUM_QUERY_
↪EXECUTORS] [-sf SCALING_FACTOR]
               [-t TIMEOUT] [-rr REQUEST_RAM] [-rc REQUEST_CPU] [-rct REQUEST_CPU_TYPE]␣
↪[-rg REQUEST_GPU]
               [-rgt REQUEST_GPU_TYPE] [-rst {None,,local-hdd,shared}] [-rss REQUEST_
↪STORAGE_SIZE]
               [-rnn REQUEST_NODE_NAME]
               {profiling,run,start,load}

Perform TPC-H inspired benchmarks in a Kubernetes cluster. This either profiles the␣
↪imported data in several DBMS and
compares some statistics, or runs the TPC-H queries. Optionally monitoring is actived.␣
↪User can choose to detach the
componenten of the benchmarking system, so that as much as possible is run inside a␣
↪Kubernetes (K8s) cluster. User can
also choose some parameters like number of runs per query and configuration and request␣
↪some resources.

positional arguments:
  {profiling,run,start,load}
                        profile the import of TPC-H data, or run the TPC-H queries, or␣
↪start DBMS and load data, or
                        just start the DBMS

optional arguments:
  -h, --help            show this help message and exit
  -db, --debug          dump debug informations
  -c CONNECTION, --connection CONNECTION
                        name of DBMS
  -cx CONTEXT, --context CONTEXT
                        context of Kubernetes (for a multi cluster environment), default␣
↪is current context
  -e EXPERIMENT, --experiment EXPERIMENT
                        sets experiment code for continuing started experiment
  -d, --detached        puts most of the experiment workflow inside the cluster
  -m, --monitoring      activates monitoring
```

(continues on next page)

(continued from previous page)

```
  -ms MAX_SUT, --max-sut MAX_SUT
                        maximum number of parallel DBMS configurations, default is no␣
↪limit
  -dt, --datatransfer   activates datatransfer
  -md MONITORING_DELAY, --monitoring-delay MONITORING_DELAY
                        time to wait [s] before execution of the runs of a query
  -nr NUM_RUN, --num-run NUM_RUN
                        number of runs per query
  -nc NUM_CONFIG, --num-config NUM_CONFIG
                        number of runs per configuration
  -ne NUM_QUERY_EXECUTORS, --num-query-executors NUM_QUERY_EXECUTORS
                        comma separated list of number of parallel clients
  -sf SCALING_FACTOR, --scaling-factor SCALING_FACTOR
                        scaling factor (SF)
  -t TIMEOUT, --timeout TIMEOUT
                        timeout for a run of a query
  -rr REQUEST_RAM, --request-ram REQUEST_RAM
                        request ram
  -rc REQUEST_CPU, --request-cpu REQUEST_CPU
                        request cpus
  -rct REQUEST_CPU_TYPE, --request-cpu-type REQUEST_CPU_TYPE
                        request node having node label cpu=
  -rg REQUEST_GPU, --request-gpu REQUEST_GPU
                        request number of gpus
  -rgt REQUEST_GPU_TYPE, --request-gpu-type REQUEST_GPU_TYPE
                        request node having node label gpu=
  -rst {None,,local-hdd,shared}, --request-storage-type {None,,local-hdd,shared}
                        request persistent storage of certain type
  -rss REQUEST_STORAGE_SIZE, --request-storage-size REQUEST_STORAGE_SIZE
                        request persistent storage of certain size
  -rnn REQUEST_NODE_NAME, --request-node-name REQUEST_NODE_NAME
                        request a specific node
```

The hardware requirements are set via

```
# pick hardware
cpu = str(args.request_cpu)
memory = str(args.request_ram)
cpu_type = str(args.request_cpu_type)
```

1.1.3 Evaluate Results in Dashboard

Evaluation is done using DBMSBenchmarker: https://github.com/Beuth-Erdelt/DBMS-Benchmarker/blob/master/docs/Dashboard.html

1.2 Concepts

An **experiment** is a benchmark of a DBMS in a certain **host setting** and a specific **benchmark setting**.

A **host setting** consists of

- an instance (a virtual machine)
- a DBMS (as a docker image)
- a volume (containing some data)
- an init script (telling the dbms how to store the data)

A **benchmark setting** consists of

- a number of client processes
- a number of runs per connection
- a maximum timeout
- a lot more, depending on the benchmark tool

1.2.1 Workflow

The **management** roughly means

- configure, set up and start a virtual machine environment
- start a DBMS and load raw data
- run some benchmarks, fetch metrics and do reporting
- shut down environment and clean up

In more detail this means

1. **Prepare Experiment**
    1. **Start Virtual Machine** AWS: Start Instance EC2 k8s: Create Deployment
    2. **Attach Network** AWS: Attach EIP k8s: Create Service, Port Forwarding
    3. **Attach Data Storage Volume** AWS: Attach and Mount EBS k8s: Attach PVC
    4. **Start Monitoring** Start Prometheus Exporter Docker Container
2. **Start Experiment**
    1. Start DBMS Docker Container Upload and run Init Scripts Load Data from Data Storage Volume
3. **Run Benchmarks**
4. **Report**
    1. **Pull Logs** From DBMS Container
    2. **Pull Metrics** From Grafana Monitoring Server

5. **Stop Experiment** AWS: Stop DBMS Docker Container, Remove Docker Remnants
6. **Clean Experiment** AWS: Unmount and Detach EBS Volume, Detach EIP, Stop Instance EC2 k8s: Stop Port Forwarding, Delete Deployment and Services

1.2.2 Prerequisits

This tool relies on

- dbms benchmarker for the actual benchmarks
- a *configuration file*
- boto for AWS management
- paramiko for SSH handling
- scp for SCP handling
- kubernetes for k8s management
- and some more python libraries

1.3 How to configure an experiment setup

We need

- a *config file* containing cluster information , say `cluster.config`
- a config folder, say `experiments/tpch/`, containing
  - a config file `queries.config` for the workload
  - folders for DDL scripts (per DBMS)
- a python script managing the experimental workflow, say `tpch.py`, see example

To use the predefined examples you will only have to change the context and namespace of the Kubernetes cluster - see below.

1.3.1 Cluster-Config

The configuration of the cluster, that is the possible host and experiment settings, consists of these parts (see also example config file):

1.3.1.1 Config of the Benchmark tool

You probably can leave this as is.

```
'benchmarker': {
    'resultfolder': './',                                   # Local path to results folder␣
↪of benchmark tool
    'jarfolder': './jars/'                                  # Optional: Local path to JDBC␣
↪drivers
},
```

- `resultfolder`: Where the benchmarker puts it's result folders

- `jarfolder`: Where the benchmarker expects the JDBC jar files

Both folders are used to correspond with the Docker containers of the benchmarker and must match the settings inside of the image.

1.3.1.2 Credentials of the Cluster

You will have to adjust the name of the namespace `my_namespace`. The rest probably can stay as is.

```
'credentials': {
    'k8s': {
        'appname': 'bexhoma',                               # K8s: To find corresponding␣
↪deployments etc
        'context': {
            'my_context': {                                 # K8s: Name of context of cluster
                'namespace': 'my_namespace',                # K8s: Namespace of User
                'clustername': 'my_cluster',                # K8s: Name of Cluster (just for␣
↪annotation)
                'service_sut': '{service}.{namespace}.svc.cluster.local',
                'port': 9091,                               # K8s: Local port for connecting␣
↪via JDBC after port forwarding
            },
        },
```

- `my_context`: Context (name) of the cluster. Repeat this section for every K8s cluster you want to use. This also allows to useand compare several Clouds.
- `namespace`: Namespace in the cluster.
- `port`: Arbitrary (free) local port that is used to ping running DBMS.

1.3.1.3 (Hardware) Monitoring

This defines where to scrape the Prometheus metrics and is defined per context (cluster). The services and the monitoring pods can be installed automatically by bexhoma.

```
'monitor': {
    'service_monitoring': 'http://{service}.{namespace}.svc.cluster.local:9090/api/v1/',
    'extend': 20,
    'shift': 0,
```

- `extend`: Number of seconds the scraping interval should be extended (at both ends).
- `shift`: Number of seconds the scraping interval should be shifted (into future).

(Hardware) Metrics

It follows a dict of hardware metrics that should be collected per DBMS. The attributes are set by bexhoms automatically so that corresponding pods can be identified. The host is found using the service of the DBMS.

```
    'metrics': {
        'total_cpu_memory': {
            'query': 'container_memory_working_set_bytes{{job="monitor-node", container_
↪label_io_kubernetes_container_name="dbms"}}/1024/1024',
            'title': 'CPU Memory [MiB]'
        },
        'total_cpu_memory_cached': {
            'query': 'container_memory_usage_bytes{{job="monitor-node", container_label_
↪io_kubernetes_container_name="dbms"}}/1024/1024',
            'title': 'CPU Memory Cached [MiB]'
        },
        'total_cpu_util': {
            'query': 'sum(irate(container_cpu_usage_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name="dbms"}}[1m]))',
            'title': 'CPU Util [%]'
        },
        'total_cpu_throttled': {
            'query': 'sum(irate(container_cpu_cfs_throttled_seconds_total{{job="monitor-
↪node", container_label_io_kubernetes_container_name="dbms"}}[1m]))',
            'title': 'CPU Throttle [%]'
        },
        'total_cpu_util_others': {
            'query': 'sum(irate(container_cpu_usage_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name!="dbms",id!="/"}}[1m]))',
            'title': 'CPU Util Others [%]'
        },
        'total_cpu_util_s': {
            'query': 'sum(container_cpu_usage_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name="dbms"}})',
            'title': 'CPU Util [s]'
        },
        'total_cpu_util_user_s': {
            'query': 'sum(container_cpu_user_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name="dbms"}})',
            'title': 'CPU Util User [s]'
        },
        'total_cpu_util_sys_s': {
            'query': 'sum(container_cpu_system_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name="dbms"}})',
            'title': 'CPU Util Sys [s]'
        },
        'total_cpu_throttled_s': {
            'query': 'sum(container_cpu_cfs_throttled_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name="dbms"}})',
            'title': 'CPU Throttle [s]'
        },
        'total_cpu_util_others_s': {
            'query': 'sum(container_cpu_usage_seconds_total{{job="monitor-node",␣
```

(continues on next page)

(continued from previous page)

```
↪container_label_io_kubernetes_container_name!="dbms",id!="/"}})',
            'title': 'CPU Util Others [s]'
        },
        'total_network_rx': {
            'query': 'sum(container_network_receive_bytes_total{{container_label_app=
↪"bexhoma", job="monitor-node"}})/1024/1024',
            'title': 'Net Rx [MiB]'
        },
        'total_network_tx': {
            'query': 'sum(container_network_transmit_bytes_total{{container_label_app=
↪"bexhoma", job="monitor-node"}})/1024/1024',
            'title': 'Net Tx [MiB]'
        },
        'total_fs_read': {
            'query': 'sum(container_fs_reads_bytes_total{{job="monitor-node", container_
↪label_io_kubernetes_container_name="dbms"}})/1024/1024',
            'title': 'FS Read [MiB]'
        },
        'total_fs_write': {
            'query': 'sum(container_fs_writes_bytes_total{{job="monitor-node", container_
↪label_io_kubernetes_container_name="dbms"}})/1024/1024',
            'title': 'FS Write [MiB]'
        },
        'total_gpu_util': {
            'query': 'sum(DCGM_FI_DEV_GPU_UTIL{{UUID=~"{gpuid}"}})',
            'title': 'GPU Util [%]'
        },
        'total_gpu_power': {
            'query': 'sum(DCGM_FI_DEV_POWER_USAGE{{UUID=~"{gpuid}"}})',
            'title': 'GPU Power Usage [W]'
        },
        'total_gpu_memory': {
            'query': 'sum(DCGM_FI_DEV_FB_USED{{UUID=~"{gpuid}"}})',
            'title': 'GPU Memory [MiB]'
        },
    }
        },
    }
},
```

1.3.1.4 Data Sources

Data sources and imports can be adressed using a key. This is organized as follows:

```
'volumes': {
    'tpch': {
        'initscripts': {
            'SF1': [
                'initschema-tpch.sql',
                'initdata-tpch-SF1.sql',
                'initdata-tpch-SF1.sh'
```

(continues on next page)

(continued from previous page)

```
            ],
            'SF1-index': [
                'initschema-tpch.sql',
                'initdata-tpch-SF1.sql',
                'initdata-tpch-SF1.sh',
                'initindexes-tpch.sql',
            ],
            'SF10': [
                'initschema-tpch.sql',
                'initdata-tpch-SF10.sql',
                'initdata-tpch-SF10.sh'
            ],
            'SF10-index': [
                'initschema-tpch.sql',
                'initdata-tpch-SF10.sql',
                'initdata-tpch-SF10.sh',
                'initindexes-tpch.sql',
            ],
        }
    }
},
```

- `tpch`: Name of the data source.
- `initscripts`: Dict of scripts to load the data source into a database. It consists of
  - a name, for example `SF1-index`,
  - a list of script names. The scripts `.sql` are sent to the command line tool of the DBMS (`loadData` - see below) and the files `.sh` are executed as shell scripts. The scripts must be present in a config folder, say `experiments/tpch/`.

The data itself must reside on a persistent volume within the cluster, that will be mounted into the DBMS container. The examples above refer to `/data/tpch/SF1/` for example.

1.3.1.5 DBMS

DBMS can be adressed using a key. We have to define some data per key, for example for the key `MonetDB` we use:

```
'dockers': {
   'MonetDB': {
        'loadData': 'cd /home/monetdb;mclient demo < {scriptname}',
        'template': {
            'version': '11.37.11',
            'alias': 'Columnwise',
            'docker_alias': 'Columnwise',
             'JDBC': {
                'auth': ['monetdb', 'monetdb'],
                'driver': 'nl.cwi.monetdb.jdbc.MonetDriver',
                'jar': 'monetdb-jdbc-2.29.jar',
                'url': 'jdbc:monetdb://{serverip}:9091/demo?so_timeout=0'
            }
        },
```

(continues on next page)

(continued from previous page)

```
        'logfile': '/tmp/monetdb5/dbfarm/merovingian.log',
        'datadir': '/var/monetdb5/',
        'priceperhourdollar': 0.0,
    },
}
```

This includes

- `loadData`: A command to run a script inside of the DBMS. This will run inside of the container of the DBMS and is used to load data. `{scriptname}` is a placeholder for the script name inside the container.
- *template*: [DBMS](https://dbmsbenchmarker.readthedocs.io/en/latest/DBMS.html) JDBC connection info that will be handed over to the benchmarker, c.f. example. Some of the data in the reference, like `hostsystem`, will be added by bexhoma automatically. The JDBC driver jar must be locally available inside the container. Some placeholders in the URL are: `serverip` (set automatically to match the corresponding pod), `dbname`, `DBNAME`, `timout_s`, `timeout_ms` (name of the database in lower and upper case, timeout in seconds and miliseconds)
- `logfile` and `datadir` that contain information about where the DBMS stores logs and databases resp.
- an optional `priceperhourdollar` that is currently ignored.

1.4 DBMS

To include a DBMS in a Kubernetes-based experiment you will need

- a Docker Image
- a JDBC Driver
- a Kubernetes Deployment Template
- some configuration
  - How to load data (DDL command)
  - DDL scripts
  - How to connect via JDBC

This document contains examples for

- *MariaDB*
- *MonetDB*
- *OmniSci*
- *PostgreSQL*

1.4.1 Example Explained

1.4.1.1 Deployment

See documentation of deployments.

1.4.1.2 Configuration

```
'dockers': {
    'OmniSci': {
        'loadData': 'bin/omnisql -u admin -pHyperInteractive < {scriptname}',     #␣
↪DBMS: Command to Login and Run Scripts
        'template': {                                                             #␣
↪Template for Benchmark Tool
            'version': 'CE v5.4',
            'alias': 'GPU',
            'docker_alias': 'GPU',
            'JDBC': {
                'driver': 'com.omnisci.jdbc.OmniSciDriver',
                'url': 'jdbc:omnisci:{serverip}:9091:omnisci',
                'auth': {'user': 'admin', 'password': 'HyperInteractive'},
                'jar': './omnisci-jdbc-4.7.1.jar'                                   #␣
↪DBMS: Local Path to JDBC Jar
            }
        },
        'logfile': '/omnisci-storage/data/mapd_log/omnisci_server.INFO',          #␣
↪DBMS: Path to Log File on Server
        'datadir': '/omnisci-storage/data/mapd_data/',                            #␣
↪DBMS: Path to directory containing data storage
        'priceperhourdollar': 0.0,                                                #␣
↪DBMS: Price per hour in USD if DBMS is rented
    }
}
```

This has

- a base name for the DBMS
- a placeholder `template` for the benchmark tool
- the JDBC driver jar locally available
- a command `loadData` for running the init scripts with `{scriptname}` as a placeholder for the script name inside the container
- `{serverip}` as a placeholder for the host address (localhost for k8s, an Elastic IP for AWS)
- `{dbname}` as a placeholder for the db name
- an optional `priceperhourdollar`
- an optional name of a `logfile` that is downloaded after the benchmark
- name of the `datadir` of the DBMS. It's size is measured using `du` after data loading has been finished.

1.4.2 MariaDB

Deployment

https://github.com/Beuth-Erdelt/Benchmark-Experiment-Host-Manager/blob/master/k8s/deploymenttemplate-MariaDB.yml

Configuration

```
'MariaDB': {
    'loadData': 'mysql < {scriptname}',
    'template': {
        'version': 'v10.4.6',
        'alias': 'GP A',
        'docker_alias': 'GP A',
        'dialect': 'MySQL',
        'JDBC': {
            'driver': "org.mariadb.jdbc.Driver",
            'auth': ["root", ""],
            'url': 'jdbc:mysql://{serverip}:9091/{dbname}',
            'jar': './mariadb-java-client-2.3.0.jar'
        }
    },
    'logfile': '',
    'datadir': '/var/lib/mysql/',
    'priceperhourdollar': 0.0,
},
```

DDL Scripts

Example for TPC-H

1.4.3 MonetDB

Deployment

https://github.com/Beuth-Erdelt/Benchmark-Experiment-Host-Manager/blob/master/k8s/deploymenttemplate-MonetDB.yml

Configuration

```
'MonetDB': {
    'loadData': 'cd /home/monetdb;mclient db < {scriptname}',
    'template': {
        'version': 'v11.31.7',
        'alias': 'In-Memory C',
        'docker_alias': 'In-Memory C',
         'JDBC': {
            'auth': ['monetdb', 'monetdb'],
            'driver': 'nl.cwi.monetdb.jdbc.MonetDriver',
            'jar': './monetdb-jdbc-2.29.jar',
            'url': 'jdbc:monetdb://{serverip}:9091/db'
        }
    },
    'logfile': '',
```

(continues on next page)

(continued from previous page)

```
    'datadir': '/var/monetdb5/',
    'priceperhourdollar': 0.0,
},
```

DDL Scripts

Example for TPC-H

1.4.4 OmniSci

Deployment

https://github.com/Beuth-Erdelt/Benchmark-Experiment-Host-Manager/blob/master/k8s/deploymenttemplate-OmniSci.yml

Configuration

```
'OmniSci': {
    'loadData': 'bin/omnisql -u admin -pHyperInteractive < {scriptname}',
    'template': {
        'version': 'CE v4.7',
        'alias': 'GPU A',
        'docker_alias': 'GPU A',
        'JDBC': {
            'driver': 'com.omnisci.jdbc.OmniSciDriver',
            'url': 'jdbc:omnisci:{serverip}:9091:omnisci',
            'auth': {'user': 'admin', 'password': 'HyperInteractive'},
            'jar': './omnisci-jdbc-4.7.1.jar'
        }
    },
    'logfile': '/omnisci-storage/data/mapd_log/omnisci_server.INFO',
    'datadir': '/omnisci-storage/',
    'priceperhourdollar': 0.0,
},
```

DDL Scripts

Example for TPC-H

1.4.5 PostgreSQL

Deployment

https://github.com/Beuth-Erdelt/Benchmark-Experiment-Host-Manager/blob/master/k8s/deploymenttemplate-PostgreSQL.yml

Configuration

```
'PostgreSQL': {
    'loadData': 'psql -U postgres < {scriptname}',
    'template': {
        'version': 'v11.4',
        'alias': 'GP D',
        'docker_alias': 'GP D',
```

(continues on next page)

(continued from previous page)

```
        'JDBC': {
            'driver': "org.postgresql.Driver",
            'auth': ["postgres", ""],
            'url': 'jdbc:postgresql://{serverip}:9091/postgres',
            'jar': './postgresql-42.2.5.jar'
        }
    },
    'logfile': '',
    'datadir': '/var/lib/postgresql/data/',
    'priceperhourdollar': 0.0,
},
```

DDL Scripts

Example for TPC-H

1.5 Deployments in Kubernetes

The deployment is expected to be given as a file named `'deployment-'+docker+'-'+instance+'.yml'` Such files are generated from a template.

Content of this document:

- How do *templates* work
  - What templates are *included*
  - Adjust the templates to *your cluster*
- How to *parametrize* templates at runtime

1.5.1 Templates

Template files are named `'deploymenttemplate-"+docker+".yml'`.

To generate a file `'deployment-'+docker+'-'+instance+'.yml'` from this

- the instance name is understood as `cpu-mem-gpu-gputype`
- the yaml file is changed as

```
dep['spec']['template']['spec']['containers'][0]['resources']['requests']['cpu'] =␣
↪cpu
dep['spec']['template']['spec']['containers'][0]['resources']['limits']['cpu'] = cpu
dep['spec']['template']['spec']['containers'][0]['resources']['requests']['memory']␣
↪= mem
dep['spec']['template']['spec']['containers'][0]['resources']['limits']['memory'] =␣
↪mem
dep['spec']['template']['spec']['nodeSelector']['gpu'] = gputype
dep['spec']['template']['spec']['containers'][0]['resources']['limits']['nvidia.com/
↪gpu'] = int(gpu)
```

1.5.1.1 Included

This repository includes some templates at https://github.com/Beuth-Erdelt/Benchmark-Experiment-Host-Manager/tree/master/k8s

DBMS included are:

- MariaDB (10.4.6)
- MonetDB (11.31.7)
- OmniSci (v5.4.0)
- PostgreSQL (11.4)

To be added in near future:

- Exasol (7.0.0) You will need a Docker image including EXAplus
- MemSQL (centos-7.1.8-43a12901be-2.0.0-1.7.0) You will have to add a licence key
- MySQL (8.0.21) You will need a Docker image including tar
- Oracle DB (18.4.0-xe) You will need to build the Docker image
- MS SQL Server (2019-CU5-ubuntu-18.04)

1.5.1.2 Your Cluster

To make these work, you may have to add the name of your Docker pull secret and the name of your persistant volume.

The default name of the secret is `private-registry-auth`:

```
imagePullSecrets:
- {name: private-registry-auth}
```

The default name of the PV is `volume-benchmarking`:

```
- name: benchmark-data-volume
  persistentVolumeClaim: {claimName: volume-benchmarking}
```

1.5.2 Parametrize Templates

The resources (requests, limits and nodeSelector) can also be set explicitly using

```
cluster.set_resources(
  requests = {
    'cpu': cpu,
    'memory': mem
  },
  limits = {
    'cpu': 0,     # unlimited
    'memory': 0   # unlimited
  },
  nodeSelector = {
    'cpu': cpu_type,
    'gpu': gpu_type,
  })
```

For further information and option see the documentation.

1.6 Monitoring

To include monitoring you will need

- a Prometheus server scraping a fixed IP / Port
- a Grafana server collecting metrics from the Prometheus server
- some *configuration* what metrics to collect

This document contains information about the

- *Concept*
- *Installation*
- *Configuration*

1.6.1 Concept

There is

- an Experiment Host - this needs Prometheus exporters
- a Monitor - this needs a Prometheus server and a Grafana server scraping the Experiment Host
- a Manager - this needs a configuration (which metrics to collect and where from)

1.6.2 Installation

To be documented

1.6.2.1 Kubernetes

- Experiment Host: Exporters are part of the deployments
- Monitor: Servers are deployed using Docker images, fixed on a separate monitoring instance
- Manager: See *configuration*

1.6.2.2 AWS

- Experiment Host: Exporters are deployed using Docker images, fixed on the benchmarked instance
- Monitor: Servers are deployed using Docker images, fixed on a separate monitoring instance
- Manager: See *configuration*

1.6.3 Configuration

We insert information about

- the Grafana server
  - access token
  - URL
- the collection
  - extension of measure intervals
  - time shift
- metrics definitions

into the cluster configuration. This is handed over to the DBMS configuration of the benchmarker in a monitoring section.

1.6.3.1 Example

The details of the metrics correspond to the YAML configuration of the deployments:

- `job="monitor-node"`
- `container_name="dbms"`

```
'monitor': {
    'grafanatoken': 'Bearer ABCDE==',
    'grafanaurl': 'http://localhost:3000/api/datasources/proxy/1/api/v1/',
    'grafanaextend': 20,
    'grafanashift': 0,
    'prometheus_url': 'http://localhost:9090/api/v1/',
    'metrics': {
        'total_cpu_memory': {
            'query': 'container_memory_working_set_bytes{{job="monitor-node", container_
↪label_io_kubernetes_container_name="dbms"}}',
            'title': 'CPU Memory [MiB]'
        },
        'total_cpu_memory_cached': {
            'query': 'container_memory_usage_bytes{{job="monitor-node", container_label_
↪io_kubernetes_container_name="dbms"}}',
            'title': 'CPU Memory Cached [MiB]'
        },
        'total_cpu_util': {
            'query': 'sum(irate(container_cpu_usage_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name="dbms"}}[1m]))',
            'title': 'CPU Util [%]'
        },
        'total_cpu_throttled': {
            'query': 'sum(irate(container_cpu_cfs_throttled_seconds_total{{job="monitor-
↪node", container_label_io_kubernetes_container_name="dbms"}}[1m]))',
            'title': 'CPU Throttle [%]'
        },
        'total_cpu_util_others': {
            'query': 'sum(irate(container_cpu_usage_seconds_total{{job="monitor-node",␣
```

(continues on next page)

(continued from previous page)

```
↪container_label_io_kubernetes_container_name!="dbms",id!="/"}}[1m]))',
            'title': 'CPU Util Others [%]'
        },
        'total_cpu_util_s': {
            'query': 'sum(container_cpu_usage_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name="dbms"}})',
            'title': 'CPU Util [s]'
        },
        'total_cpu_throttled_s': {
            'query': 'sum(container_cpu_cfs_throttled_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name="dbms"}})',
            'title': 'CPU Throttle [s]'
        },
        'total_cpu_util_others_s': {
            'query': 'sum(container_cpu_usage_seconds_total{{job="monitor-node",␣
↪container_label_io_kubernetes_container_name!="dbms",id!="/"}})',
            'title': 'CPU Util Others [s]'
        },
        'total_network_rx': {
            'query': 'sum(container_network_receive_bytes_total{{container_label_app=
↪"dbmsbenchmarker", job="monitor-node"}})',
            'title': 'Net Rx [b]'
        },
        'total_network_tx': {
            'query': 'sum(container_network_transmit_bytes_total{{container_label_app=
↪"dbmsbenchmarker", job="monitor-node"}})',
            'title': 'Net Tx [b]'
        },
        'total_fs_read': {
            'query': 'sum(container_fs_reads_bytes_total{{job="monitor-node", container_
↪label_io_kubernetes_container_name="dbms"}})',
            'title': 'FS Read [b]'
        },
        'total_fs_write': {
            'query': 'sum(container_fs_writes_bytes_total{{job="monitor-node", container_
↪label_io_kubernetes_container_name="dbms"}})',
            'title': 'FS Write [b]'
        },
        'total_gpu_util': {
            'query': 'sum(DCGM_FI_DEV_GPU_UTIL{{UUID=~"{gpuid}"}})',
            'title': 'GPU Util [%]'
        },
        'total_gpu_power': {
            'query': 'sum(DCGM_FI_DEV_POWER_USAGE{{UUID=~"{gpuid}"}})',
            'title': 'GPU Power Usage [W]'
        },
        'total_gpu_memory': {
            'query': 'sum(DCGM_FI_DEV_FB_USED{{UUID=~"{gpuid}"}})',
            'title': 'GPU Memory [MiB]'
        },
    }
}
```

Fine Tuning

If the Grafana server has metrics coming from general Prometheus server, that is it scrapes more exporters than just the bexhoma related, we will need to specify further which metrics we are interested in.

There is a placeholder `{gpuid}` that is substituted automatically by a list of GPUs present in the pod.

1.7 Contributing to Bexhoma

You would like to contribute? Great!

Some things that you can help on include:

- **New Workloads**: The `example/` folder includes the TPC-H and TPC-DS (reading) queries in various dialects. We are interested in adding other relevant workloads.
- **Documentation**: If a point in the documentation is unclear, we look forward to receiving tips and suggestions for improvement.
- **Testing**: If the behavior is not as expected and you suspect a bug, please report it to our issue tracker.
- **Use Cases**: If you have had any experiences with peculiarities, mistakes, ambiguities or oddities or particularly beautiful cases etc., we are interested in hearing about them and passing them on to others.

1.7.1 Non-code contributions

Even if you don't feel ready or able to contribute code, you can still help out. There always things that can be improved on the documentation (even just proof reading, or telling us if a section isn't clear enough).

1.7.2 Code contributions

We welcome pull requests to fix bugs or add new features.

1.7.2.1 Licensing

In your git commit and/or pull request, please explicitly state that you agree to your contributions being licensed under "GNU Affero General Public License v3".

1.7.2.2 Git Usage

If you are planning to make a pull request, start by creating a new branch with a short but descriptive name (rather than using your master branch).

1.7.2.3 Coding Conventions

DBMSBenchmarker tries to follow the coding conventions laid out in PEP8 and PEP257

- http://www.python.org/dev/peps/pep-0008/
- http://www.python.org/dev/peps/pep-0257/

1.7.2.4 Testing

New features or functions will not be accepted without testing. Likewise for any enhancement or bug fix, we require including an additional test.

If the feature or functionality concerns benchmarking

We expect it to be testable with the TPC-H workload. Otherwise please add references to data and queries you used to test the functionality. They must be publicly accessible.

If the feature or functionality concerns evaluation

Please include a zipped result folder, so we can trace the enhancement of evaluations based on the same results you have used. (And please be sure to not include secret passwords in the connection information).

1.8 bexhoma package API

1.8.1 Subpackages

1.8.1.1 bexhoma.scripts package

Submodules

bexhoma.scripts.experimentsmanager module

This script contains some code snippets for testing the detached mode in Kubernetes

`bexhoma.scripts.experimentsmanager.manage()`

bexhoma.scripts.tpcds module

Date
2021-02-12

Version
0.1

Authors
Patrick Erdelt

Perform TPC-DS inspired benchmarks in a Kubernetes cluster. This either profiles the imported data in several DBMS and compares some statistics, or runs the TPC-H queries. Optionally monitoring is actived. User can choose to detach the componenten of the benchmarking system, so that as much as possible is run inside a Kubernetes (K8s) cluster. User can also choose some parameters like number of runs per query and configuration and request some resources.

bexhoma.scripts.tpcds.**do_benchmark**()

bexhoma.scripts.tpch module

Date
2021-02-12

Version
0.1

Authors
Patrick Erdelt

Perform TPC-H inspired benchmarks in a Kubernetes cluster. This either profiles the imported data in several DBMS and compares some statistics, or runs the TPC-H queries. Optionally monitoring is actived. User can choose to detach the componenten of the benchmarking system, so that as much as possible is run inside a Kubernetes (K8s) cluster. User can also choose some parameters like number of runs per query and configuration and request some resources.

bexhoma.scripts.tpch.**do_benchmark**()

Module contents

The clustermanager module

1.8.2 Submodules

1.8.2.1 bexhoma.clusters module

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Module to manage testbeds. Historically this supported different implementations based on IaaS. All methods will be deprecated except for Kubernetes (K8s), so the structure will change in future.

class bexhoma.clusters.**aws**(*clusterconfig='cluster.config'*, *experiments_configfolder='experiments/'*, *yamlfolder='k8s/'*, *context=None*, *code=None*, *instance=None*, *volume=None*, *docker=None*, *script=None*, *queryfile=None*)

Bases: *kubernetes*

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Class for containing Kubernetes methods specific to AWS. This adds handling of nodegroups for elasticity.

check_nodegroup(*nodegroup_type=''*, *nodegroup_name=''*, *num_nodes_aux_planned=0*)

eksctl(*command*)

Runs an eksctl command.

Parameters
command – An eksctl command

Returns
stdout of the eksctl command

get_nodegroup_size(*nodegroup_type=''*, *nodegroup_name=''*)

get_nodes(*app=''*, *nodegroup_type=''*, *nodegroup_name=''*)

Get all nodes of a cluster. This overwrites the cluster method with the AWS specific nodegroup-name label.

Parameters

- **app** – Name of the pod

- **nodegroup_type** – Type of the nodegroup, e.g. sut
- **nodegroup_name** – Name of the nodegroup, e.g. sut_high_memory

scale_nodegroup(*nodegroup_name*, *size*)

scale_nodegroups(*nodegroup_names*, *size=None*)

wait_for_nodegroup(*nodegroup_type=''*, *nodegroup_name=''*, *num_nodes_aux_planned=0*)

wait_for_nodegroups(*nodegroup_names*, *size=None*)

class bexhoma.clusters.**kubernetes**(*clusterconfig='cluster.config'*, *experiments_configfolder='experiments/'*, *yamlfolder='k8s/'*, *context=None*, *code=None*, *instance=None*, *volume=None*, *docker=None*, *script=None*, *queryfile=None*)

Bases: *testbed*

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Class for containing specific Kubernetes (K8s) methods. This class can be overloaded to define specific implementations of Kubernetes, for example AWS.

add_experiment(*experiment*)

Add an experiment to this cluster.

Parameters
experiment – Experiment object

store_pod_log(*pod_name*, *container=''*)

Store the log of a pod in a local file in the experiment result folder. Optionally the name of a container can be given (mandatory, if pod has multiple containers).

Parameters

- **pod_name** – Name of the pod
- **container** – Name of the container

class bexhoma.clusters.**testbed**(*clusterconfig='cluster.config'*, *experiments_configfolder='experiments/'*, *yamlfolder='k8s/'*, *context=None*, *code=None*, *instance=None*, *volume=None*, *docker=None*, *script=None*, *queryfile=None*)

Bases: object

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Class to manage experiments in a Kubernetes cluster.

TODO:

- Remove instance / volume references from IaaS
- Documentation for purpose and position
- Documentation for "copy log and init" mechanisms
- Clearify if *OLD_* can be reused

`OLD__getTimediff`()

`OLD_continueBenchmarks`(*connection=None*, *query=None*)

`OLD_getChildProcesses`()

`OLD_runReporting`()

`OLD_startPortforwarding`(*service=''*, *app=''*, *component='sut'*)

`OLD_stopPortforwarding`()

`add_to_messagequeue`(*queue*, *data*)
Add data to (Redis) message queue.

Parameters
- **queue** – Name of the queue
- **data** – Data to be added to queue

`check_DBMS_connection`(*ip*, *port*)
Check if DBMS is open for connections. Tries to open a socket to ip:port. Returns True if this is possible.

Parameters
- **ip** – IP of the host to connect to
- **port** – Port of the server on the host to connect to

Returns
True, iff connecting is possible

cluster_access()

provide access to an K8s cluster by initializing connection handlers.

connect_dashboard()

Connects to the dashboard component. This means the output ports of the dashboard component are forwarded to localhost. Expect results be available under port 8050 (dashboard) and 8888 (Jupyter).

connect_master(*experiment=''*, *configuration=''*)

Connects to the master node of a sut component. This means the output ports of the component are forwarded to localhost. Must be limited to a specific experiment or dbms configuration.

Parameters

- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

copyInits()

copyLog()

create_dashboard_name(*app=''*, *component='dashboard'*)

Creates a suitable name for the dashboard component.

Parameters

- **app** – app the dashboard belongs to
- **component** – Component name, should be 'dashboard' typically

dashboard_is_running()

Returns True, iff dashboard is running.

Returns
True, iff dashboard is running

delay(*sec*)

Function for waiting some time and inform via output about this. Synonymous for wait()

Parameters
sec – Number of seconds to wait

delete_deployment(*deployment*)

Delete a deployment given by name.

Parameters
deployment – Name of the deployment to be deleted.

delete_job(*jobname=''*, *app=''*, *component=''*, *experiment=''*, *configuration=''*, *client=''*)

Delete a job given by name or matching a set of labels (component/ experiment/ configuration)

Parameters

- **jobname** – Name of the job we want to delete
- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – DEPRECATED?

delete_job_pods(*jobname=''*, *app=''*, *component=''*, *experiment=''*, *configuration=''*, *client=''*)

Delete all pods of a job given by name or matching a set of labels (component/ experiment/ configuration)

Parameters

- **jobname** – Name of the job we want to delete the pods of
- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – DEPRECATED?

delete_pod(*name*)

Delete a pod given by name

Parameters
name – name of the pod to be deleted

delete_pvc(*name*)

Delete a persistent volume claim given by name

Parameters
name – name of the pvc to be deleted

delete_service(*name*)

Delete a service given by name

Parameters
name – name of the service to be deleted

delete_stateful_set(*name*)

Delete a stateful set given by name

Parameters
name – name of the stateful set to be deleted

downloadLog()

execute_command_in_pod(*command*, *pod=''*, *container=''*, *params=''*)

Runs an shell command remotely inside a container of a pod.

Parameters

- **command** – A shell command
- **pod** – The name of the pod
- **container** – The name of the container in the pod
- **params** – Optional parameters, currently ignored

Returns
stdout of the shell command

get_dashboard_pod_name(*app=''*, *component='dashboard'*)

Returns the name of the dashboard pod.

Parameters

- **app** – app the dashboard belongs to

- **component** – Component name, should be 'dashboard' typically

Returns
name of the dashboard pod

get_deployments(*app=''*, *component=''*, *experiment=''*, *configuration=''*)
Return all deployments matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the deployment belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

get_job_pods(*app=''*, *component=''*, *experiment=''*, *configuration=''*, *client=''*)
Return all pods of a jobs matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – DEPRECATED?

get_job_status(*jobname=''*, *app=''*, *component=''*, *experiment=''*, *configuration=''*, *client=''*)
Return status of a jobs given by name or matching a set of labels (component/ experiment/ configuration)

Parameters

- **jobname** – Name of the job we want to know the status of
- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – DEPRECATED?

get_jobs(*app=''*, *component=''*, *experiment=''*, *configuration=''*, *client=''*)
Return all jobs matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – DEPRECATED?

get_nodes(*app=''*, *nodegroup_type=''*, *nodegroup_name=''*)

Get all nodes of a cluster.

Parameters

- **app** – Name of the pod
- **nodegroup_type** – Type of the nodegroup, e.g. sut
- **nodegroup_name** – Name of the nodegroup, e.g. sut_high_memory

get_pod_status(*pod*, *app=''*)

Return status of a pod given by name

Parameters

- **app** – app the set belongs to
- **pod** – Name of the pod the status of which should be returned

get_pods(*app=''*, *component=''*, *experiment=''*, *configuration=''*, *status=''*)

Return all pods matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the pod belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **status** – Status of the pod

get_pods_labels(*app=''*, *component=''*, *experiment=''*, *configuration=''*)

Return all labels of pods matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the set belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

get_ports_of_service(*app=''*, *component=''*, *experiment=''*, *configuration=''*)

Return all ports of a services matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the service belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

get_pvc(*app=''*, *component=''*, *experiment=''*, *configuration=''*)

Return all persistent volume claims matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the pvc belongs to

- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

get_pvc_labels(*app=''*, *component=''*, *experiment=''*, *configuration=''*, *pvc=''*)

Return all labels of persistent volume claims matching a set of labels (component/ experiment/ configuration) or name

Parameters

- **app** – app the pvc belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **pvc** – Name of the PVC

get_pvc_specs(*app=''*, *component=''*, *experiment=''*, *configuration=''*, *pvc=''*)

Return all specs of persistent volume claims matching a set of labels (component/ experiment/ configuration) or name

Parameters

- **app** – app the pvc belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **pvc** – Name of the PVC

get_pvc_status(*app=''*, *component=''*, *experiment=''*, *configuration=''*, *pvc=''*)

Return status of persistent volume claims matching a set of labels (component/ experiment/ configuration) or name

Parameters

- **app** – app the pvc belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **pvc** – Name of the PVC

get_services(*app=''*, *component=''*, *experiment=''*, *configuration=''*)

Return all services matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the service belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

get_stateful_sets(*app=''*, *component=''*, *experiment=''*, *configuration=''*)

Return all stateful sets matching a set of labels (component/ experiment/ configuration)

Parameters

- **app** – app the set belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

kubectl(*command*)

Runs an kubectl command in the current context.

Parameters
command – An eksctl command

Returns
stdout of the kubectl command

log_experiment(*experiment*)

Function to log current step of experiment. This is supposed to be written on disk for comprehension and repetition. This should be reworked and yield a YAML format for example. Moreover this should respect "new" workflows with detached parallel loaders for example.

Parameters
experiment – Dict that stores parameters of current experiment stept

pod_log(*pod*, *container=''*)

restart_dashboard(*app=''*, *component='dashboard'*)

Stops the dashboard component and its service.

Parameters

- **app** – app the dashboard belongs to
- **component** – Component name, should be 'dashboard' typically

set_code(*code*)

Sets the unique identifier of an experiment. Use case: We start a cluster (without experiment), then define an experiment, which creates an identifier. This identifier will be set in the cluster as the default experiment.

Parameters
code – Unique identifier of an experiment

set_connectionmanagement(***kwargs*)

Sets connection management data for the experiments. This is for the benchmarker component (dbms-benchmarker). Can be overwritten by experiment and configuration.

Parameters
kwargs – Dict of meta data, example 'timout' => 60

set_ddl_parameters(***kwargs*)

Sets DDL parameters for the experiments. This substitutes placeholders in DDL script. Can be overwritten by experiment and configuration.

Parameters
kwargs – Dict of meta data, example 'index' => 'btree'

set_experiment(*instance=None*, *volume=None*, *docker=None*, *script=None*)

Sets a specific setting for an experiment. In particular this sets instance, volume and dbms (docker image) and name of a list of DDL scrips. This typically comes from a cluster.config.

Parameters

- **instances** – Dict of instances (DEPRECATED, was for IaaS?)
- **volumes** – Dict of volumes, that carry data
- **dockers** – Dict of docker images and meta data about how to usw
- **script** – Name of list of DDL scripts, that are run when start_loading() is called

set_experiments(*instances=None*, *volumes=None*, *dockers=None*)

Assigns dicts containing information about instances, volumes and dbms (docker images). This typically comes from a cluster.config.

Parameters

- **instances** – Dict of instances (DEPRECATED, was for IaaS?)
- **volumes** – Dict of volumes, that carry data
- **dockers** – Dict of docker images and meta data about how to usw

set_experiments_configfolder(*experiments_configfolder*)

Sets the configuration folder for the experiments. Bexhoma expects subfolders for expeiment types, for example tpch. In there, bexhoma looks for query.config files (for dbmsbenchmarker) and subfolders containing the schema per dbms.

Parameters
experiments_configfolder – Relative path to an experiment folder

set_queryfile(*queryfile*)

Sets the name of a query file of an experiment. This is for the benchmarker component (dbmsbenchmarker).

Parameters
code – Unique identifier of an experiment

set_querymanagement(***kwargs*)

Sets query management data for the experiments. This is for the benchmarker component (dbmsbenchmarker).

Parameters
kwargs – Dict of meta data, example 'numRun' => 3

set_resources(***kwargs*)

Sets resources for the experiments. This is for the SUT component. Can be overwritten by experiment and configuration.

Parameters
kwargs – Dict of meta data, example 'requests' => {'cpu' => 4}

set_workload(***kwargs*)

Sets mata data about the experiments for example name and description.

Parameters
kwargs – Dict of meta data, example 'name' => 'TPC-H'

start_dashboard(*app=''*, *component='dashboard'*)

Starts the dashboard component and its service, if there is no such pod. Manifest is expected in 'deploymenttemplate-bexhoma-dashboard.yml'.

Parameters

- **app** – app the dashboard belongs to
- **component** – Component name, should be 'dashboard' typically

start_messagequeue(*app=''*, *component='messagequeue'*)

Starts the message queue. Manifest is expected in 'deploymenttemplate-bexhoma-messagequeue.yml'

Parameters

- **app** – app the messagequeue belongs to
- **component** – Component name, should be 'messagequeue' typically

stop_benchmarker(*experiment=''*, *configuration=''*)

Stops all benchmarking components (jobs and their pods) in the cluster. Can be limited to a specific experiment or dbms configuration.

Parameters

- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

stop_dashboard(*app=''*, *component='dashboard'*)

Stops the dashboard component and its service.

Parameters

- **app** – app the dashboard belongs to
- **component** – Component name, should be 'dashboard' typically

stop_loading(*experiment=''*, *configuration=''*)

Stops all loading components (jobs and their pods) in the cluster. Can be limited to a specific experiment or dbms configuration.

Parameters

- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

stop_maintaining(*experiment=''*, *configuration=''*)

Stops all maintaining components (jobs and their pods) in the cluster. Can be limited to a specific experiment or dbms configuration.

Parameters

- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

stop_monitoring(*app=''*, *component='monitoring'*, *experiment=''*, *configuration=''*)

Stops all monitoring components (deployments and their pods) in the cluster and their service. Can be limited to a specific experiment or dbms configuration.

Parameters

- **experiment** – Unique identifier of the experiment

- **configuration** – Name of the dbms configuration

stop_sut(*app=''*, *component='sut'*, *experiment=''*, *configuration=''*)

Stops all sut components (deployments and their pods, stateful sets and services) in the cluster. Can be limited to a specific experiment or dbms configuration.

Parameters

- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

wait(*sec*)

Function for waiting some time and inform via output about this

Parameters
sec – Number of seconds to wait

1.8.2.2 bexhoma.configurations module

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Class for managing an DBMS configuation. This is plugged into an experiment object.

class bexhoma.configurations.**default**(*experiment*, *docker=None*, *configuration=''*, *script=None*, *alias=None*, *num_experiment_to_apply=None*, *clients=[1]*, *dialect=''*, *worker=0*, *dockerimage=''*)

Bases: object

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Class for managing an DBMS configuation. This is plugged into an experiment object.

param experiment
Unique identifier of the experiment

param docker
Name of the Docker image

param configuration
Name of the configuration

param script
Unique identifier of the experiment

param alias
Unique identifier of the experiment

`OLD_get_items`(*app=''*, *component=''*, *experiment=''*, *configuration=''*)

`add_benchmark_list`(*list_clients*)

Add a list of (number of) benchmarker instances, that are to benchmark the current SUT. Example: *[1,2,1]* means sequentially we will have 1, then 2 and then 1 benchmarker instances.

Parameters
list_clients – List of (number of) benchmarker instances

`attach_worker`()

Attaches worker nodes to the master of the sut. This runs the dockertemplate['attachWorker'] command.

`check_DBMS_connection`(*ip*, *port*)

Check if DBMS is open for connections. Tries to open a socket to ip:port. Returns True if this is possible.

Parameters

- **ip** – IP of the host to connect to
- **port** – Port of the server on the host to connect to

Returns
True, iff connecting is possible

`check_load_data`()

Check if loading of the sut has finished. If yes, store timeLoadingStart, timeLoadingEnd, timeLoading as labels and in this object as attributes. If there is a loading job: check if all pods are completed. Copy the logs of the containers in the pods and remove the pods. If there is no loading job: Read the labels. If loaded is True, store the timings in this object as attributes.

`check_sut`()

Check if the pod of the sut is running. If yes, store it's name in *self.pod_sut*.

`client`

If we have a sequence of benchmarkers, this tells at which position we are

configuration

Name of the configuration, default: Name of the Docker image

connection_parameter

Collect all parameters that might be interesting in evaluation of results

copyLog()

create_manifest_benchmarking(*connection*, *app=''*, *component='benchmarker'*, *experiment=''*, *configuration=''*, *experimentRun=''*, *client='1'*, *parallelism=1*, *alias=''*, *env={}*, *template=''*, *num_pods=1*)

Creates a job template for the benchmarker. This sets meta data in the template and ENV.

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – Number of benchmarker if there is a sequence of benchmarkers
- **parallelism** – Number of parallel pods in job
- **alias** – Alias name of the dbms
- **env** – Dict of environment variables
- **template** – Template for the job manifest

Returns

Name of file in YAML format containing the benchmarker job

create_manifest_job(*app=''*, *component='benchmarker'*, *experiment=''*, *configuration=''*, *experimentRun=''*, *client='1'*, *parallelism=1*, *env={}*, *template=''*, *nodegroup=''*, *num_pods=1*)

Creates a job and sets labels (component/ experiment/ configuration).

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **experimentRun** – Number of run of the configuration in this experiment
- **client** – Number of client of the job for this experiment run
- **parallelism** – Number of parallel pods in this job
- **env** – Dict of environment variables for the job manifest
- **template** – Template name of the job manifest
- **nodegroup** – Nodegroup of the pods of the job
- **num_pods** – Number of pods that run in total

create_manifest_loading(*app=''*, *component='loading'*, *experiment=''*, *configuration=''*, *parallelism=1*, *alias=''*, *num_pods=1*)

Creates a job template for loading. This sets meta data in the template and ENV.

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **parallelism** – Number of parallel pods in job

Returns
Name of file in YAML format containing the loading job

create_manifest_maintaining(*app=''*, *component='maintaining'*, *experiment=''*, *configuration=''*, *parallelism=1*, *alias=''*, *num_pods=1*)

Creates a job template for maintaining. This sets meta data in the template and ENV.

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **parallelism** – Number of parallel pods in job

Returns
Name of file in YAML format containing the maintaining job

create_monitoring(*app=''*, *component='monitoring'*, *experiment=''*, *configuration=''*)

Generate a name for the monitoring component. Basically this is *{app}-{component}-{configuration}-{experiment}-{client}*

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

delay(*sec*)

Function for waiting some time and inform via output about this. Synonymous for wait()

Parameters
sec – Number of seconds to wait

docker

Name of the Docker image

dockerimage

Name of the Docker image of the SUT

dockertemplate

Template of the Docker information taken from cluster.config

execute_command_in_pod_sut(*command*, *pod=''*, *container='dbms'*, *params=''*)

Runs an shell command remotely inside a container of a pod. This defaults to the current sut's pod and the container "dbms"

Parameters

- **command** – A shell command
- **pod** – The name of the pod - default current sut's pod
- **container** – The name of the container in the pod - default dbms
- **params** – Optional parameters, currently ignored

Returns

stdout of the shell command

experiment

Unique identifier of the experiment

experiment_done

True, iff the SUT has performed the experiment completely

generate_component_name(*app=''*, *component=''*, *experiment=''*, *configuration=''*, *experimentRun=''*, *client=''*)

Generate a name for the component. Basically this is *{app}-{component}-{configuration}-{experiment}-{client}*

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – Number of the client, if it comes from a sequence of same components (in particular benchmarker)

getTimediff()

get_connection_config(*connection*, *alias=''*, *dialect=''*, *serverip='localhost'*, *monitoring_host='localhost'*)

Returns information about the sut's host disk space. Basically this calls *df /* on the host.

Returns

Size of disk in Bytes

get_host_all()

Calls all *get_host_x()* methods. Returns information about the sut's host as a dict.

Returns

Dict of informations about the host

get_host_cores()

Returns information about the sut's host CPU's cores. Basically this calls *grep -c ^processor /proc/cpuinfo* on the host.

Returns
CPU's cores of the host

`get_host_cpu()`

Returns information about the sut's host CPU. Basically this calls *more /proc/cpuinfo | grep 'model name'* on the host.

Returns
CPU of the host

`get_host_cuda()`

Returns information about the sut's host CUDA version. Basically this calls *nvidia-smi | grep 'CUDA'* on the host.

Returns
CUDA version of the host

`get_host_diskspace_used()`

Returns information about the sut's host disk space. Basically this calls *df /* on the host.

Returns
Size of disk in Bytes

`get_host_diskspace_used_data()`

Returns information about the sut's host disk space used for the data (the database). Basically this calls *du* on the host directory that is mentioned in cluster.config as to store the database.

Returns
Size of disk used for database in Bytes

`get_host_gpu_ids()`

Returns information about the sut's host GPUs. Basically this calls *nvidia-smi -L* on the host and generates a list of UUIDs of the GPUs.

Returns
List of GPU UUIDs of the host

`get_host_gpus()`

Returns information about the sut's host GPUs. Basically this calls *nvidia-smi -L* on the host and aggregates result.

Returns
GPUs of the host

`get_host_memory()`

Returns information about the sut's host RAM. Basically this calls *grep MemTotal /proc/meminfo* on the host.

Returns
RAM of the host

`get_host_node()`

Returns information about the sut's host name. Basically this calls *kubectl get pod* to receive the information.

Returns
Node name of the host

`get_host_system()`

Returns information about the sut's host OS. Basically this calls *uname -r* on the host.

Returns
OS of the host

`get_instance_from_resources()`

Generates an instance name out of the resource parameters that are set using *set_resources()*. Should be DEPRECATED and replaced by something better.

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

`load_data()`

Start loading data into the sut. This runs *load_data_asynch()* as an asynchronous thread. At first *prepare_init_dbms()* is run.

`loading_after_time`

Time as an integer when initial loading should start - to give the system time to start up completely

`loading_finished`

Time as an integer when initial loading has finished

`loading_started`

Time as an integer when initial loading has started

`maintaining_is_pending()`

Returns True, iff maintaining is in pending state.

Returns
True, if maintaining is in pendig state

`maintaining_is_running()`

Returns True, iff maintaining is running.

Returns
True, if dbms is running

`monitoring_is_pending()`

Returns True, iff monitoring is in pending state.

Returns
True, if monitoring is in pendig state

`monitoring_is_running()`

Returns True, iff monitoring is running.

Returns
True, if monitoring is running

`pod_sut`

Name of the sut's master pod

`prepare_init_dbms()`

Prepares to load data into the dbms. This copies the DDL scripts to /tmp on the host of the sut. Optionally parameters in DDL script are replaced by ddl_parameters. The files are renamed *filled_* then.

reset_sut()

Forget that the SUT has been loaded and benchmarked.

run_benchmarker_pod(*connection=None*, *alias=''*, *dialect=''*, *query=None*, *app=''*, *component='benchmarker'*, *experiment=''*, *configuration=''*, *client='1'*, *parallelism=1*)

Runs the benchmarker job. Sets meta data in the connection.config. Copy query.config and connection.config to the first pod of the job (result folder mounted into every pod)

Parameters

- **connection** – Name of configuration prolonged by number of runs of the sut (installations) and number of client in a sequence of
- **alias** – An alias can be given if we want to anonymize the dbms
- **dialect** – A name of a SQL dialect can be given
- **query** – The benchmark can be fixed to a specific query
- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – Number of benchmarker this is in a sequence of
- **parallelism** – Number of parallel benchmarker pods we want to have

set_connectionmanagement(***kwargs*)

Sets connection management data for the experiment. This is for the benchmarker component (dbmsbenchmarker). Can be overwritten by configuration.

Parameters
kwargs – Dict of meta data, example 'timout' => 60

set_ddl_parameters(***kwargs*)

Sets DDL parameters for the experiments. This substitutes placeholders in DDL script. Can be overwritten by configuration.

Parameters
kwargs – Dict of meta data, example 'index' => 'btree'

set_eval_parameters(***kwargs*)

Sets some arbitrary parameters that are supposed to be handed over to the benchmarker component. Can be set by experiment before creation of configuration.

Parameters
kwargs – Dict of meta data, example 'type' => 'noindex'

set_experiment(*instance=None*, *volume=None*, *docker=None*, *script=None*)

Read experiment details from cluster config

set_loading(*parallel*, *num_pods=None*)

Sets job parameters for loading components: Number of parallel pods and optionally (if different) total number of pods. By default total number of pods is set to number of parallel pods. Can be set by experiment before creation of configuration.

Parameters

- **parallel** – Number of parallel pods
- **num_pods** – Optionally (if different) total number of pods

set_loading_parameters(***kwargs*)

Sets ENV for loading components. Can be set by experiment before creation of configuration.

Parameters
kwargs – Dict of meta data, example 'PARALLEL' => '64'

set_maintaining(*parallel*, *num_pods=None*)

Sets job parameters for maintaining components: Number of parallel pods and optionally (if different) total number of pods. By default total number of pods is set to number of parallel pods. Can be set by experiment before creation of configuration.

Parameters

- **parallel** – Number of parallel pods
- **num_pods** – Optionally (if different) total number of pods

set_maintaining_parameters(***kwargs*)

Sets ENV for maintaining components. Can be set by experiment before creation of configuration.

Parameters
kwargs – Dict of meta data, example 'PARALLEL' => '64'

set_nodes(***kwargs*)

set_resources(***kwargs*)

Sets resources for the experiment. This is for the SUT component. Can be overwritten by experiment and configuration.

Parameters
kwargs – Dict of meta data, example 'requests' => {'cpu' => 4}

set_storage(***kwargs*)

Sets parameters for the storage that might be attached to components. This is in particular for the database of dbms under test. Example:

storageClassName = 'ssd', storageSize = '100Gi', keep = False

Can be set by experiment before creation of configuration.

Parameters
kwargs – Dict of meta data, example 'storageSize' => '100Gi'

start_loading(*delay=0*)

Starts data ingestion by calling scripts inside the sut (dbms) container.

Parameters
delay – Number of seconds to wait after calling scripts

start_loading_pod(*app=''*, *component='loading'*, *experiment=''*, *configuration=''*, *parallelism=1*, *num_pods=1*)

Starts a job for parallel data ingestion.

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment

- **configuration** – Name of the dbms configuration
- **parallelism** – Number of parallel pods in job

start_maintaining(*app=''*, *component='maintaining'*, *experiment=''*, *configuration=''*, *parallelism=1*, *num_pods=1*)

Starts a maintaining job.

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **parallelism** – Number of parallel pods in job

start_monitoring(*app=''*, *component='monitoring'*, *experiment=''*, *configuration=''*)

Starts a monitoring deployment.

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

start_sut(*app=''*, *component='sut'*, *experiment=''*, *configuration=''*)

Start the system-under-test (dbms). This also controls optional worker and storage. Resources are set according to *set_resources()*.

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

stop_loading(*app=''*, *component='loading'*, *experiment=''*, *configuration=''*)

Stops a loading job and removes all its pods.

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

stop_maintaining(*app=''*, *component='maintaining'*, *experiment=''*, *configuration=''*)

Stops a monitoring deployment and removes all its pods.

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring

- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

stop_monitoring(*app=''*, *component='monitoring'*, *experiment=''*, *configuration=''*)

Stops a monitoring deployment and removes its service.

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

stop_sut(*app=''*, *component='sut'*, *experiment=''*, *configuration=''*)

Stops a sut deployment and removes all its services and (optionally) stateful sets. It also stops and removes all related components (monitoring, maintaining, loading).

Parameters

- **app** – app the component belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration

sut_is_pending()

Returns True, iff system-under-test (dbms) is in pending state.

Returns
True, if dbms is in pendig state

sut_is_running()

Returns True, iff system-under-test (dbms) is running.

Returns
True, if dbms is running

timeLoading

Time in seconds the system has taken for the initial loading of data

use_storage()

Return True, iff storage for the database should be used. Otherwise database is inside ephemeral-storage.

wait(*sec*, *silent=False*)

Function for waiting some time and inform via output about this

Parameters

- **sec** – Number of seconds to wait
- **silent** – True means we do not output anything about this waiting

class bexhoma.configurations.**hammerdb**(*experiment*, *docker=None*, *configuration=''*, *script=None*, *alias=None*, *num_experiment_to_apply=None*, *clients=[1]*, *dialect=''*, *worker=0*, *dockerimage=''*)

Bases: *default*

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Class for managing an DBMS configuation. This is plugged into an experiment object.

param experiment
Unique identifier of the experiment

param docker
Name of the Docker image

param configuration
Name of the configuration

param script
Unique identifier of the experiment

param alias
Unique identifier of the experiment

`create_manifest_benchmarking`(*connection*, *app=''*, *component='benchmarker'*, *experiment=''*, *configuration=''*, *experimentRun=''*, *client='1'*, *parallelism=1*, *alias=''*, *num_pods=1*)

Creates a job template for the benchmarker. This sets meta data in the template and ENV. This sets some settings specific to HammerDB.

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – Number of benchmarker if there is a sequence of benchmarkers
- **parallelism** – Number of parallel pods in job
- **alias** – Alias name of the dbms

Returns
Name of file in YAML format containing the benchmarker job

`run_benchmarker_pod`(*connection=None*, *alias=''*, *dialect=''*, *query=None*, *app=''*, *component='benchmarker'*, *experiment=''*, *configuration=''*, *client='1'*, *parallelism=1*)

Runs the benchmarker job. Sets meta data in the connection.config. Copy query.config and connection.config to the first pod of the job (result folder mounted into every pod)

Parameters

- **connection** – Name of configuration prolonged by number of runs of the sut (installations) and number of client in a sequence of
- **alias** – An alias can be given if we want to anonymize the dbms
- **dialect** – A name of a SQL dialect can be given
- **query** – The benchmark can be fixed to a specific query
- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – Number of benchmarker this is in a sequence of
- **parallelism** – Number of parallel benchmarker pods we want to have

bexhoma.configurations.**load_data_asynch**(*app*, *component*, *experiment*, *configuration*, *pod_sut*, *scriptfolder*, *commands*, *loadData*, *path*, *volume*, *context*)

class bexhoma.configurations.**ycsb**(*experiment*, *docker=None*, *configuration=''*, *script=None*, *alias=None*, *num_experiment_to_apply=None*, *clients=[1]*, *dialect=''*, *worker=0*, *dockerimage=''*)

Bases: *default*

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Class for managing an DBMS configuation. This is plugged into an experiment object.

param experiment
Unique identifier of the experiment

param docker
Name of the Docker image

param configuration
Name of the configuration

param script
Unique identifier of the experiment

param alias
Unique identifier of the experiment

create_manifest_benchmarking(*connection*, *app=''*, *component='benchmarker'*, *experiment=''*, *configuration=''*, *experimentRun=''*, *client='1'*, *parallelism=1*, *alias=''*, *num_pods=1*)

Creates a job template for the benchmarker. This sets meta data in the template and ENV. This sets some settings specific to YCSB.

Parameters

- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – Number of benchmarker if there is a sequence of benchmarkers
- **parallelism** – Number of parallel pods in job
- **alias** – Alias name of the dbms

Returns

Name of file in YAML format containing the benchmarker job

run_benchmarker_pod(*connection=None*, *alias=''*, *dialect=''*, *query=None*, *app=''*, *component='benchmarker'*, *experiment=''*, *configuration=''*, *client='1'*, *parallelism=1*)

Runs the benchmarker job. Sets meta data in the connection.config. Copy query.config and connection.config to the first pod of the job (result folder mounted into every pod)

Parameters

- **connection** – Name of configuration prolonged by number of runs of the sut (installations) and number of client in a sequence of
- **alias** – An alias can be given if we want to anonymize the dbms
- **dialect** – A name of a SQL dialect can be given
- **query** – The benchmark can be fixed to a specific query
- **app** – app the job belongs to
- **component** – Component, for example sut or monitoring
- **experiment** – Unique identifier of the experiment
- **configuration** – Name of the dbms configuration
- **client** – Number of benchmarker this is in a sequence of
- **parallelism** – Number of parallel benchmarker pods we want to have

1.8.2.3 bexhoma.experiments module

Date
2022-10-01

Version
0.6.0

Authors
Patrick K. Erdelt

Classes for managing an experiment. This is plugged into a cluster object. It collects some configuation objects. Two examples are included, dealing with TPC-H and TPC-DS tests. Another example concerns TSBS experiment. Each experiment also should have an own folder having:

- a query file
- a subfolder for each dbms, that may run this experiment, including schema files

class bexhoma.experiments.**DictToObject**(*dictionary*)

Bases: object

https://coderwall.com/p/idfiea/python-dict-to-object

class bexhoma.experiments.**default**(*cluster*, *code=None*, *num_experiment_to_apply=1*, *timeout=7200*, *detached=True*)

Bases: object

Class for defining an experiment. Settings are set generally. This class should be overloaded to define specific experiments.

add_benchmark_list(*list_clients*)

Add a list of (number of) benchmarker instances, that are to benchmark the current SUT. Example *[1,2,1]* means sequentially we will have 1, then 2 and then 1 benchmarker instances. This is applied to all dbms configurations of the experiment.

Parameters
list_clients – List of (number of) benchmarker instances

add_configuration(*configuration*)

Adds a configuration object to the list of configurations of this experiment. When a new configuration object is instanciated, an experiment object has to be provided. This method is then called automatically.

Parameters
configuration – Configuration object

benchmark_list(*list_clients*)

DEPRECATED? Is not used anymore. Runs a given list of benchmarker applied to all running SUTs of experiment.

Parameters
list_clients – List of (number of) benchmarker instances

delay(*sec*)

Function for waiting some time and inform via output about this. Synonymous for wait()

Parameters
sec – Number of seconds to wait

end_benchmarking(*jobname*)

Ends a benchmarker job. This is for storing or cleaning measures.

Parameters
jobname – Name of the job to clean

end_loading(*jobname*)

Ends a loading job. This is for storing or cleaning measures.

Parameters
jobname – Name of the job to clean

evaluate_results(*pod_dashboard=''*)

Let the dashboard pod build the evaluations. This is specific to dbmsbenchmarker.

1) All local logs are copied to the pod.
2) Benchmarker in the dashboard pod is updated (dev channel)
3) All results of all DBMS are joined (merge.py of benchmarker) in dashboard pod
4) Evaluation cube is built (python benchmark.py read -e yes) in dashboard pod

set_connectionmanagement(***kwargs*)

Sets connection management data for the experiment. This is for the benchmarker component (dbmsbenchmarker). Can be overwritten by configuration.

Parameters
kwargs – Dict of meta data, example 'timout' => 60

set_ddl_parameters(***kwargs*)

Sets DDL parameters for the experiments. This substitutes placeholders in DDL script. Can be overwritten by configuration.

Parameters
kwargs – Dict of meta data, example 'index' => 'btree'

set_eval_parameters(***kwargs*)

Sets some arbitrary parameters that are supposed to be handed over to the benchmarker component. Can be overwritten by configuration.

Parameters
kwargs – Dict of meta data, example 'type' => 'noindex'

set_experiment(*instance=None*, *volume=None*, *docker=None*, *script=None*)

Read experiment details from cluster config

Parameters

- **instance** –
- **volume** –
- **docker** –

- **script** –

set_experiments_configfolder(*experiments_configfolder*)

Sets the configuration folder for the experiment. Bexhoma expects subfolders for expeiment types, for example tpch. In there, bexhoma looks for query.config files (for dbmsbenchmarker) and subfolders containing the schema per dbms.

Parameters
experiments_configfolder – Relative path to an experiment folder

set_loading(*parallel*, *num_pods=None*)

Sets job parameters for loading components: Number of parallel pods and optionally (if different) total number of pods. By default total number of pods is set to number of parallel pods. Can be overwritten by configuration.

Parameters

- **parallel** – Number of parallel pods
- **num_pods** – Optionally (if different) total number of pods

set_loading_parameters(***kwargs*)

Sets ENV for loading components. Can be overwritten by configuration.

Parameters
kwargs – Dict of meta data, example 'PARALLEL' => '64'

set_maintaining(*parallel*, *num_pods=None*)

Sets job parameters for maintaining components: Number of parallel pods and optionally (if different) total number of pods. By default total number of pods is set to number of parallel pods. Can be overwritten by configuration.

Parameters

- **parallel** – Number of parallel pods
- **num_pods** – Optionally (if different) total number of pods

set_maintaining_parameters(***kwargs*)

Sets ENV for maintaining components. Can be overwritten by configuration.

Parameters
kwargs – Dict of meta data, example 'PARALLEL' => '64'

set_nodes(***kwargs*)

set_queryfile(*queryfile*)

Sets the name of a query file of the experiment. This is for the benchmarker component (dbmsbenchmarker).

Parameters
code – Unique identifier of an experiment

set_querymanagement(***kwargs*)

Sets query management data for the experiment. This is for the benchmarker component (dbmsbenchmarker).

Parameters
kwargs – Dict of meta data, example 'numRun' => 3

set_querymanagement_monitoring(*numRun=256*, *delay=10*, *datatransfer=False*)

Sets some parameters that are supposed to be suitable for a monitoring test:

- high number of runs
- optional delay
- optional data transfer
- monitoring active

Parameters

- **numRun** – Number of runs per query (this is for the benchmarker component)
- **delay** – Number of seconds to wait between queries (this is for the benchmarker component)
- **datatransfer** – If data should we retrieved and compared

set_querymanagement_quicktest(*numRun=1*, *datatransfer=False*)

Sets some parameters that are supposed to be suitable for a quick functional test:

- small number of runs
- no delay
- optional data transfer
- no monitoring

Parameters

- **numRun** – Number of runs per query (this is for the benchmarker component)
- **datatransfer** – If data should we retrieved and compared

set_resources(***kwargs*)

Sets resources for the experiment. This is for the SUT component. Can be overwritten by experiment and configuration.

Parameters
kwargs – Dict of meta data, example 'requests' => {'cpu' => 4}

set_storage(***kwargs*)

Sets parameters for the storage that might be attached to components. This is in particular for the database of dbms under test. Example:

storageClassName = 'ssd', storageSize = '100Gi', keep = False

Can be overwritten by configuration.

Parameters
kwargs – Dict of meta data, example 'storageSize' => '100Gi'

set_workload(***kwargs*)

Sets mata data about the experiment, for example name and description.

Parameters
kwargs – Dict of meta data, example 'name' => 'TPC-H'

start_loading()

Tells all dbms configurations of this experiment to start loading data.

start_monitoring()

Start monitoring for all dbms configurations of this experiment.

start_sut()

Start all dbms configurations of this experiment.

stop_benchmarker(*configuration=''*)

Stop all benchmarker jobs of this experiment. If a dbms configurations is given, use it. Otherwise tell the cluster to stop all benchmarker jobs belonging to this experiment code.

stop_loading()

Stop all loading jobs of this experiment. If a list of dbms configurations is set, use it. Otherwise tell the cluster to stop all loading jobs belonging to this experiment code.

stop_maintaining()

Stop all maintaining jobs of this experiment. If a list of dbms configurations is set, use it. Otherwise tell the cluster to stop all maintaining jobs belonging to this experiment code.

stop_monitoring()

Stop all monitoring deployments of this experiment. If a list of dbms configurations is set, use it. Otherwise tell the cluster to stop all monitoring deployments belonging to this experiment code.

stop_sut()

Stop all SUT deployments of this experiment. If a list of dbms configurations is set, use it. Otherwise tell the cluster to stop all monitoring deployments belonging to this experiment code.

test_results()

Run test script in dashboard pod. Extract exit code.

Returns
exit code of test script

wait(*sec*)

Function for waiting some time and inform via output about this

Parameters
sec – Number of seconds to wait

work_benchmark_list(*intervals=30*, *stop=True*)

Run typical workflow:

1) start SUT
2) start monitoring
3) start loading (at first scripts (schema or loading via pull), then optionally parallel loading pods)
4) optionally start maintaining pods
5) at the same time as 4. run benchmarker jobs corresponding to list given via add_benchmark_list()

Parameters

- **intervals** – Seconds to wait before checking change of status
- **stop** – Tells if SUT should be removed when all benchmarking has finished. Set to False if we want to have loaded SUTs for inspection.

zip()

Zip the result folder in the dashboard pod.

class bexhoma.experiments.**iot**(*cluster*, *code=None*, *queryfile='queries-iot.config'*, *SF='1'*, *num_experiment_to_apply=1*, *timeout=7200*)

Bases: *default*

Class for defining an TSBS experiment. This sets

- the folder to the experiment - including query file and schema informations per dbms
- name and information about the experiment
- additional parameters - here SF (the scaling factor)

set_queries_full()

set_queries_profiling()

set_querymanagement_maintaining(*numRun=128*, *delay=5*, *datatransfer=False*)

class bexhoma.experiments.**tpcc**(*cluster*, *code=None*, *SF='1'*, *num_experiment_to_apply=1*, *timeout=7200*)

Bases: *default*

Class for defining an TPC-C experiment (in the HammerDB version). This sets

- the folder to the experiment - including query file and schema informations per dbms
- name and information about the experiment
- additional parameters - here SF (the scaling factor), i.e. number of warehouses

end_benchmarking(*jobname*)

Ends a benchmarker job. This is for storing or cleaning measures.

Parameters
jobname – Name of the job to clean

test_results()

Run test script locally. Extract exit code.

Returns
exit code of test script

class bexhoma.experiments.**tpcds**(*cluster*, *code=None*, *queryfile='queries-tpcds.config'*, *SF='100'*, *num_experiment_to_apply=1*, *timeout=7200*)

Bases: *default*

Class for defining an TPC-DS experiment. This sets

- the folder to the experiment - including query file and schema informations per dbms
- name and information about the experiment
- additional parameters - here SF (the scaling factor)

set_queries_full()

set_queries_profiling()

class bexhoma.experiments.**tpch**(*cluster*, *code=None*, *queryfile='queries-tpch.config'*, *SF='100'*, *num_experiment_to_apply=1*, *timeout=7200*)

Bases: *default*

Class for defining an TPC-H experiment. This sets

- the folder to the experiment - including query file and schema informations per dbms
- name and information about the experiment
- additional parameters - here SF (the scaling factor)

set_queries_full()

set_queries_profiling()

class bexhoma.experiments.**tsbs**(*cluster*, *code=None*, *queryfile='queries-tsbs.config'*, *SF='1'*, *num_experiment_to_apply=1*, *timeout=7200*)

Bases: *default*

Class for defining an TSBS experiment. This sets

- the folder to the experiment - including query file and schema informations per dbms
- name and information about the experiment
- additional parameters - here SF (the scaling factor)

end_loading(*jobname*)

Ends a loading job. This is for storing or cleaning measures.

Parameters
jobname – Name of the job to clean

set_queries_full()

set_queries_profiling()

set_querymanagement_maintaining(*numRun=128*, *delay=5*, *datatransfer=False*)

class bexhoma.experiments.**ycsb**(*cluster*, *code=None*, *SF='1'*, *num_experiment_to_apply=1*, *timeout=7200*)

Bases: *default*

Class for defining an YCSB experiment. This sets

- the folder to the experiment - including query file and schema informations per dbms
- name and information about the experiment
- additional parameters - here SF (the scaling factor), i.e. number of rows divided by 10.000

end_benchmarking(*jobname*)

Ends a benchmarker job. This is for storing or cleaning measures.

Parameters
jobname – Name of the job to clean

end_loading(*jobname*)

Ends a loading job. This is for storing or cleaning measures.

Parameters
jobname – Name of the job to clean

`log_to_df`(*filename*)

`test_results`()

Run test script locally. Extract exit code.

Returns
exit code of test script

1.8.3 Module contents

The clustermanager module

1.9 Benchmark Experiment Host Manager (Bexhoma)

This Python tools helps **managing benchmark experiments of Database Management Systems (DBMS) in a Kubernetes-based High-Performance-Computing (HPC) cluster environment**. It enables users to configure hardware / software setups for easily repeating tests over varying configurations.

It serves as the **orchestrator** [2] for distributed parallel benchmarking experiments in a Kubernetes Cloud. This has been tested at Amazon Web Services, Google Cloud, Microsoft Azure, IBM Cloud, Oracle Cloud, and at Minikube installations, running with Clickhouse, Exasol, Citus Data (Hyperscale), IBM DB2, MariaDB, MariaDB Columnstore, MemSQL (SingleStore), MonetDB, MySQL, OmniSci (HEAVY.AI), Oracle DB, PostgreSQL, SQL Server, SAP HANA, TimescaleDB, and Vertica. .

The basic workflow is [1,2]: start a virtual machine, install monitoring software and a database management system, import data, run benchmarks (external tool) and shut down everything with a single command. A more advanced workflow is: Plan a sequence of such experiments, run plan as a batch and join results for comparison.

See the homepage and the documentation.

If you encounter any issues, please report them to our Github issue tracker.

1.9.1 Installation

1. Download the repository: https://github.com/Beuth-Erdelt/Benchmark-Experiment-Host-Manager
2. Install the package `pip install bexhoma`
3. Make sure you have a working `kubectl` installed (Also make sure to have access to a running Kubernetes cluster - for example Minikube)
4. Adjust configuration
    1. Rename `k8s-cluster.config` to `cluster.config`
    2. Set name of context, namespace and name of cluster in that file
5. Install data [tbd in detail] Example for TPC-H SF=1:
    - Run `kubectl create -f k8s/job-data-tpch-1.yml`
    - When job is done, clean up with `kubectl delete job -l app=bexhoma -l component=data-source` and `kubectl delete deployment -l app=bexhoma -l component=data-source`.

6. Install result folder Run `kubectl create -f k8s/pvc-bexhoma-results.yml`

1.9.2 Quickstart

The repository contains a tool for running TPC-H (reading) queries at MonetDB and PostgreSQL.

1. Run `tpch run -sf 1 -t 30`.
2. You can watch status using `bexperiments status` while running. This is equivalent to `python cluster.py status`.
3. After benchmarking has finished, run `bexperiments dashboard` to connect to a dashboard. You can open dashboard in browser at `http://localhost:8050`. This is equivalent to `python cluster.py dashboard` Alternatively you can open a Jupyter notebook at `http://localhost:8888`.

1.9.3 More Informations

For full power, use this tool as an orchestrator as in [2]. It also starts a monitoring container using Prometheus and a metrics collector container using cAdvisor. It also uses the Python package dbmsbenchmarker, [3], as query executor and evaluator as in [1,2]. See the images folder for more details.

1.9.4 Contributing, Bug Reports

If you have any question or found a bug, please report them to our Github issue tracker. In any bug report, please let us know:

- Which operating system and hardware (32 bit or 64 bit) you are using
- Python version
- Bexhoma version (or git commit/date)
- Traceback that occurs (the full error message)

We are always looking for people interested in helping with code development, documentation writing, technical administration, and whatever else comes up. If you wish to contribute, please first read the contribution section or visit the documentation.

1.9.5 References

If you use Bexhoma in work contributing to a scientific publication, we kindly ask that you cite our application note [2] or [1]:

[1] A Framework for Supporting Repetition and Evaluation in the Process of Cloud-Based DBMS Performance Benchmarking

Erdelt P.K. (2021) A Framework for Supporting Repetition and Evaluation in the Process of Cloud-Based DBMS Performance Benchmarking. In: Nambiar R., Poess M. (eds) Performance Evaluation and Benchmarking. TPCTC 2020. Lecture Notes in Computer Science, vol 12752. Springer, Cham. https://doi.org/10.1007/978-3-030-84924-5_6

[2] Orchestrating DBMS Benchmarking in the Cloud with Kubernetes

Erdelt P.K. (2022) Orchestrating DBMS Benchmarking in the Cloud with Kubernetes. In: Nambiar R., Poess M. (eds) Performance Evaluation and Benchmarking. TPCTC 2021. Lecture Notes in Computer Science, vol 13169. Springer, Cham. https://doi.org/10.1007/978-3-030-94437-7_6

[3] DBMS-Benchmarker: Benchmark and Evaluate DBMS in Python

Erdelt P.K., Jestel J. (2022). DBMS-Benchmarker: Benchmark and Evaluate DBMS in Python. Journal of Open Source Software, 7(79), 4628 https://doi.org/10.21105/joss.04628

1.10 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

E

G

H

I

K

L

M

O

P

R

S

T

U

W

Y

Z