**Deep Data Bench Usage and Documentation**

**2/18/2016**

**What is Deep Data Bench?**

Deep Data Bench is a versatile benchmarking tool that runs with your data.

Rather than using a consistent schema and transaction profile, Deep Data Bench uses special meta data files and configuration files to define and run its tests. These files are highly customizable and generated from a schema you specify.

Deep Data Bench allows you to define tests that run various CRUD profiles in parallel, serial, and with different numbers of threads. It allows for a fine level of control over what those queries will look like and how long or frequently they are run.

Deep Data Bench is relatively simple to use; the more access you give it the more fine tuned and realistic your simulated data will be. If you generate your metadata from a live production database your simulated data’s cardinality, relative row size, and schema will be preserved. If you generate the metadata from a schema only database the cardinality and row size will be randomized and can be manually tuned. If you provide a slow query log your metadata will automatically match the CRUD ratios from that log file.

Once you have generated metadata describing your database Deep Data Bench provides a wide array of testing options. The tool allows you to do true A/B testing; pre-generating all it’s query data, running the exact same queries over and over again against different MySQL configurations, schemas, or hardware setups.

Because of how Deep Data Bench builds up its simulated data it also allows a wide variety of scale testing. Tests can be written in such a way that you run the same set of queries on a 100 thousand row, 1 million rows, 10 million row, 100 million row, etc. data set. Tests can also be run continuously so that data size and complexity continues to grow as the test goes on.

**What is Required to run Deep Data Bench?**

In order to run Deep Data Bench you will need a to meet the 4 requirements below.

1. A source database or metadata file.

Deep Data Bench requires a metadata file to run benchmarks. You can either point Deep Data Bench at a running database with the schema/configuration you would like to benchmark, or you can provide it a metadata file.

Metadata files can be downloaded from our samples repository or generated using the metadata.py script.

1. Python 2.7

Some of the special Data generation techniques used in Deep Data Bench require libraries not found in Python 2.6. Running Deep Data Bench with Python 2.6 will have unexpected results. Because of this Python 2.7 is required to run Deep Data Bench.

1. Python MySQLdb module

In order to communicate with a MySQL database Deep Data Bench requires the MySQLdb module. In a standard setup this can be installed via pip or your systems package manager (python-mysqldb on Ubuntu and MySQL-python on CentOS).

1. The Deep Data Bench Tooling

You will need to download all of the Deep Data Bench Tooling onto a system with enough resources and access to run your tests.

**What makes up Deep Data Bench?**

Deep Data Bench is a collection of Python scripts. Many of these scripts can be run as standalone pieces to generate the various files used by Deep Data Bench.

In the majority of cases you will only have to worry about the main deepdatabench script. The interface is interactive and will prompt you through running your test. This will eventually call and utilize the other scripts listed below. It is the main test orchestrator and will prompt for database access, generate all the necessary meta data files, generate queries (either on-the fly during testing or before running any tests), run queries, collect statistics, save a report file, and print a summary.

There is a metadata script that connects to a source database and performs queries to collect cardinality, data size, row count, and schema information.

There is a slow query parser that reads a MySQL slow query and generates a summary of CRUD types/counts per table.

There is a query generator that uses the metadata script and then generates and stores random query information that Deep Data Bench can later consume for testing.

There is a report viewer that will display a summary along with any errors, warnings, etc. from a given Deep Data Bench report file.

There are several empty directories used to store output report files, generated query files, and generated metadata files.

There are also several internal files that are not standalone. These are responsible for tasks such as metrics parsing, benchmark orchestration, and plot generation.

**What are the specifics of each Deep Data Bench File?**

**­­­­­**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Inputs** | **Outputs** | **Standalone­­­­** | **Description** |
| bigdatabench.py | Configuration (given via interactive mode, command line, my.ini file, or any combination of the above. | * Complete metadata file * Data Files * Benchmark Report | Yes | The main executable to run benchmarks |
| metadata.py | * Database access information * Slow Query log (optional) | Complete metadata file | Yes | Generates schema/database metadata |
| ParseSlowQuery.py | Slow Query Log | Partial metadata file containing CRUD profile | Yes (non-standard) | Summarizes a Slow Query Log |
| QueryGenerator.py | * Database access information * Test information (clients, length, tables, database) | DBD data files (in data directory) | Yes (non-standard) | Generates benchmark queries and transactions. |
| DBAppConductor.py | N/A | N/A | No | Internal benchmark orchestration file |
| ClientQueryStats.py | N/A | N/A | No | Internal statistics/results parsing file |
| PillarReport.py | DDB Report File |  | Yes (use ReportViewer.py instead) | View a summary of a benchmark |
| ReportViewer.py | DDB Report File | None | Yes | View warnings, errors, and summary of a benchmark |
| meta\_data | N/A | N/A | N/A | Directory where generated metadata files are stored |
| data | N/A | N/A | N/A | Directory where generated query data is stored |
| reports | N/A | N/A | N/A | Directory where Finished Benchmark Reports are stored. |

**How Do I run Deep Data Bench?**

Deep Data Bench can is run by utilizing the bigdatabench.py script. There are three ways you can define how Deep Data Bench runs a test. The easiest method is to simply run the script, it has an interactive mode that will guide your through benchmark configuration. If you would like your benchmarking to be more automated you can also specify all of the options via the command line, or put them in a configuration file and specify that configuration file using the --config flag.

For a more detailed guide see the Deep Data Bench Walkthrough.

**What does a Deep Data Bench Configuration File Look Like?**

Here is a sample bigdatabench command along with a config file that would do identical work. The config file can be named anything as long as you point to it with the --config flag.

bigdatabench.py --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_mysql\_port=3306 --source\_database=ddb\_demo\_sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_mysql\_port=3306 --destination\_database=ddb\_demo\_sysbench\_destination --tables='\*' --pillars="Pureload,sysbench, Pureload sysbench" --pillar\_durations="100000,12s, 12s" --num\_clients="4,4,2 2" --show\_stats\_frequency=30 --pre\_generate\_data=True

[MySQL]

### Source Connection Information ###

source\_mysql\_user = root

source\_mysql\_password = foobar

### Specify either a socket or a host and port.

#source\_mysql\_socket = /tmp/mysqld.sock

source\_mysql\_host = 127.0.0.1

source\_mysql\_port = 3306

source\_database = ddb\_demo\_sysbench

### Destination Connection Information ###

destination\_mysql\_user = root

destination\_mysql\_password = foobar

### Specify either a socket or a host and port.

# destination\_mysql\_socket = /tmp/mysqld.sock

destination\_mysql\_host = 127.0.0.1

destination\_mysql\_port = 3306

destination\_database = ddb\_demo\_sysbench\_destination

### Specify \* for all tables in database or choose specific tables.

tables = \*

### Set to False to start with a fresh destination database.

### Set to True to retain the destination database.

### (for when analytics are of interest and data loading takes too long, etc.).

### Warning: False will DROP and recreate the destination database.

### Note the default is False and this was omitted from the cmd line.

retain\_destination\_database = False

### Set the MySQL engine of the tables that are created.

destination\_mysql\_engine = None

### Beta feature for in-depth data collection. Keep this set to False.

### Note the default is False and this was omitted from the cmd line.

collect\_stats = False

[Pillar]

pillars = Pureload, sysbench, Pureload sysbench

### Duration for each pillar in seconds (120s) or row count (1000).

pillar\_durations = 100000,12s, 12s

#pillar\_durations = 10000000,120s, 120s

num\_clients = 4,4,2 2

### How often stats are printed during a test.

show\_stats\_frequency = 30

### Set to False to pregenerate all data at the beginning of the test.

### Set to False to generate data as necessary.

pre\_generate\_data = True

### Note the default is 1 and this was omitted from the cmd line.

repeat\_x\_times = 1

### Note the default is None and this was omitted from the cmd line.

report\_name = None

**What are the Deep Data Bench Configuration Parameters?**

The Deep Data Bench configuration file has two sections. The first is the MySQL section. This defines connection information for the source database that the simulation will be based off of. It defines connection information for the destination database that the simulation will be running on. And it contains a few flags that tell it what tables to act on and whether to retain data, and which engines to test with.

The second section is Big Data Bench specific. It allows you to specify a pillar or list of pillars (also called a profile) and then the duration for each pillar as well as the number of client to run each with. There are also a few flags that tell the test how often to print information, whether or not to generate all data before running any queries, etc.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Input Format** | **Required** | **Description** |
| tables | ‘\*’ or space separated list i.e. ‘table1 table2 table3’ | No | List of tables to act on. \* indicates all. |
| retain\_destination\_database | ‘True’ or ‘False’ | No | Setting this to false will blow away and recreate the destination database every time the test is run. Setting this to True will not recreate the destination schema. This is useful if you’d like to continually grow a dataset or if you have grown a dataset and are more interested in CRUD/Analytics. |
| destination\_mysql\_engine | Engine string i.e. ‘Deep’ or ‘InnoDB’ | No | Specifies which engine to use on schema; does not work if retain\_destination\_database is True. |
| collect\_stats | ‘True’ or ‘False’ | No | Beta feature to collect more in depth stats. |
| pillars | Comma separated list each containing a string or space separated list of strings that correspond to a default profile or metadata file. i.e. PureLoad, CRUD, /tmp/mytest.meta, Analytics CRUD. | Yes | Each pillar represents a phase of the test and a single metadata file. Any item that is a space separated list is treated as a single pillar and both profiles are run in parallel. All other pillars are run in serial. |
| pillar\_durations | Comma separated list each containing a single value. Values can be either a number of seconds or a row/transaction count i.e. 120s, 10000, 600s, 10000. \*Number of values must match number of pillars. | Yes | The length of time or number of transactions each pillar will be run for. |
| num\_clients | Comma separated list each containing a single value. Values is an int. \*Number of values must correspond to the number of pillars. If pillars contains a space separated value num\_clients must also i.e. (2, 4, 4, 2 2) | Yes | The number of clients/threads each pillar will be run at. |
| show\_stats\_frequency | Time in seconds i.e. 30 | No |  |
| pre\_generate\_data | ‘True’ or ‘False’ | No | Pregenerate data before running any queries. This should be True for any A/B benchmarking and False for any long duration/stress testing. Setting to True will make queries go a small amount faster and more consistently cross runs. |
| repeate\_x\_times | Int | No | Repeat the exact test x times. |
| report\_name | ‘None’ or a String | No | If None DDB will use a random name, if a string report files will be prefixed with that name. |

**What is a Deep Data Bench Profile and What Do They Do?**

A Deep Data Bench profile is what defines how the test will be run. This can either be a hand-tuned/pre-made metadata file or it can be one of the default Profiles built into the tool.

These profiles essentially define two things. The first is a set of tuning dials that control things how queries are generated and run. This includes things such as extended insert size, rollback chance, transaction size, range sizes, where clause/join clause definitions, savepoint frequency, etc. For a complete listing see the table below. The second is simply defining CRUD ratios, as in the below:

{ 'CRUD' : {'INSERT' : 1000, 'SELECT' : 0, 'UPDATE' : 0, 'DELETE' : 0, 'REPLACE' : 0}, 'SLEEP' : {'MIN' : 0, 'MAX' : 0}}

For a detailed list of each of the default profiles take a look at the metadata.py script (Each of them has 12+ toggles that is too much information to go over here). Note that when specifying a profile Deep Data Bench is case sensitive. Here is a quick summary:

|  |  |
| --- | --- |
| **Name** | **Description** |
| PureLoad | Do nothing but inserts with a reasonable extended insert size. |
| JustInserts | Do nothing but inserts with defaults. |
| JustUpdates | Do nothing but Updates with defaults. |
| JustDeletes | Do nothing but Deletes with defaults. |
| JustReplaces | Do nothing but Replaces with defaults. |
| Sysbench | Run a test that mimics sysbench behavior. |
| TRX\_CRUD | Run even amounts of CRUD in large transactions. |
| EvenCRUD | Run even amounts of CRUD without transactions. |
| Analytics | Run Analytics queries. |
| Random | Randomize everything. |
| RandomWithLoad | Randomize everything but ensure that data is Inserted. |

**What is a Deep Data Bench Metadata file?**

The metadata file is the file that is generated by all of the initial tooling. It stores data that is utilized for Query Generation and it can be saved off and reused to re-run tests.

When running Deep Data Bench each pillar will generate or utilize a separate metadata file. So if you are running with pillars = PureLoad, CRUD, Analytics you will end up with the PureLoad.json, CRUD.json, and Analyitcs.json in your meta\_data folder.

It contains seven sections.

The first section describes the actual data for each database.table.column. A max/min is defined for each column along with the method DDB will use for generating data for that column, the cardinality, the data type, and any Foreign Key relationships. When tuning you may want to play with the max, min, and method fields.

The second section defines the CRUD ratios for each database.table. This is in the format shown above. It also includes the total size of the table. When tuning you will want to play with all of these values.

The third section lists out all the indexes for Deep Data Bench. You should not touch this for tuning.

The fourth section contains a schema dump. While the fifth section specifically contains any MySQL views. Again you shouldn’t mess with these while tuning.

The sixth section is where all the query tuning knobs and dials are kept. You will want to play with this while tuning, see the below table.

Finaly the seventh section contains fixed data on a database.table.column basis. This is useful if you have a specific column that is categorical and should only ever have 1 of a few values (such as the status of a customer or category of a class).

Here is a sample metadata file generated by running the PureLoad profile against the sysbench database.

[

{

"sbtest": [

{

"column\_name": "id",

"current\_max": null,

"current\_min": null,

"datatype": "int",

"foreign\_keys": [],

"max": 4294967295,

"method": "autoinc",

"min": 0,

"uniqueness": 100

},

{

"column\_name": "k",

"current\_max": null,

"current\_min": null,

"datatype": "int",

"foreign\_keys": [],

"max": 4294967295,

"method": "autoinc",

"min": 0,

"uniqueness": 100

},

{

"column\_name": "c",

"current\_max": null,

"current\_min": null,

"datatype": "char",

"foreign\_keys": [],

"max": 24,

"method": "random",

"min": 12,

"uniqueness": 100

},

{

"column\_name": "pad",

"current\_max": null,

"current\_min": null,

"datatype": "char",

"foreign\_keys": [],

"max": 12,

"method": "random",

"min": 6,

"uniqueness": 100

}

]

},

{

"sbtest": {

"CRUD": {

"DELETE": 470,

"INSERT": 955,

"REPLACE": 291,

"SELECT": 160,

"UPDATE": 53

},

"SIZE": 0,

"SLEEP": {

"MAX": 0,

"MIN": 0

},

"TABLE\_TYPE": "BASE TABLE"

}

},

{

"sbtest": {

"PRIMARY": [

"id"

],

"k": [

"k"

]

}

},

{

"sbtest": "CREATE TABLE `sbtest` (\n `id` int(10) unsigned NOT NULL AUTO\_INCREMENT,\n `k` int(10) unsigned NOT NULL DEFAULT '0',\n `c` char(120) NOT NULL DEFAULT '',\n `pad` char(60) NOT NULL DEFAULT '',\n PRIMARY KEY (`id`),\n KEY `k` (`k`)\n) ENGINE=InnoDB DEFAULT CHARSET=latin1"

},

{},

{

"commit\_early\_chance": 26,

"create\_savepoint\_chance": 38,

"delete\_limit\_size": 10,

"extended\_insert\_size": 35,

"max\_tables\_in\_a\_transaction": 256,

"max\_transaction\_size": 200,

"min\_transaction\_size": 7,

"pre\_generate\_data": false,

"range\_max\_date\_days": 64,

"range\_max\_float": 840,

"range\_max\_int": 452,

"range\_max\_time\_minutes": 6,

"range\_max\_year": 1,

"release\_savepoint\_chance": 19,

"rollback\_chance": 32,

"rollback\_early\_chance": 14,

"rollback\_to\_savepoint\_chance": 5,

"scale\_factor": 1,

"select\_group\_by\_chance": 8,

"select\_join\_chance": 2,

"select\_limit\_size": 1000,

"select\_lock\_in\_share\_mode\_chance": 23,

"select\_order\_by\_chance": 3,

"update\_limit\_size": 10,

"update\_primary\_key\_columns": true,

"where\_clause\_early\_termination\_chance": 3,

"where\_clause\_range\_chance": 22

},

{

"table1.column2": [

"gg",

"ff",

"\*"

]

}

**What are the Deep Data Bench Metadata Parameters?**

Here is a list of all the parameters in the metadata file and what they do.

Note that anything specified as a chance is a 1/X chance of that event occurring, where 0 is never. For example setting select\_join\_chance to 10 will give a 1/10 chance (10%) for any generated select query to contain a join.

These values can all be changed in your metadata file. It may also be possible to tweak the metadata.py script to generate more favorable automated metadata profiles.

}

|  |  |
| --- | --- |
| **Name** | **Description** |
| current\_max | The recorded max when the metadata was generated. Do not change this. |
| current\_min | The recorded min when the metadata was generated. Do not change this. |
| datatype | The data type of the column. For a complete listing of valid values see the metadata.py script. These correspond to MySQL types. |
| foreign\_keys | A list of table.column FK relationships that must be honored. |
| max | The max value that can be generated. |
| method | The method DDB should use for data generation. Options are random, autoinc, and ignore. |
| min | The min value that can be generated. |
| uniqueness | The original cardinality of this column. |

|  |  |
| --- | --- |
| **Name** | **Description** |
| CRUD | The CRUD ratios to use. |
| SIZE | The original size of the table. DDB will try to keep these values proportional to the original. |
| SLEEP MAX | The maximum time a sleep thread should sleep. (0 for never sleep) |
| SLEEP MIN | The minimum time a sleep thread should sleep. |
| TABLE\_TYPE | The MySQL table type (VIEW or BASE TABLE) |

|  |  |
| --- | --- |
| **Name** | **Description** |
| commit\_early\_chance | Chance to commit before transaction size is reached. |
| create\_savepoint\_chance | Chance to create a savepoint when in a transaction. |
| delete\_limit\_size | Maximum number of rows to delete in a single command. |
| extended\_insert\_size | Extended Insert Size for all Inserts. |
| max\_tables\_in\_a\_transaction | Max number of tables in a single transaction. |
| max\_transaction\_size | Max queries in a transaction. |
| min\_transaction\_size | Minimum queries in a transaction. |
| range\_max\_date\_days | Largest number of days used when generating range queries. |
| range\_max\_float | Largest difference used when generating float range queries. |
| pre\_generate\_data | Whether or not to pregenerate data. |
| range\_max\_int | Largest difference used when generating int range queries. |
| range\_max\_time\_minutes | Largest number of minutes used when generating range queries. |
| range\_max\_year | Largest number of years used when generating range queries. |
| release\_savepoint\_chance | Chance a savepoint will be released during a transaction. |
| rollback\_chance | Chance a rollback will be done at end of transaction. |
| rollback\_early\_chance | Chance a rollback will be done during a transaction. |
| rollback\_to\_savepoint\_chance | Chance a rollback will go to savepoint. |
| scale\_factor | When doing a simulated scale test, scale up by this much. |
| select\_group\_by\_chance | Chance for a group by clause during queries. |
| select\_join\_chance | Chance for a join to occur during queries. |
| select\_limit\_size | Limit used for all Selects. |
| select\_lock\_in\_share\_mode\_chance | Chance to lock in share mode during selects. |
| select\_order\_by\_chance | Chance a order by clause will be used during selects. |
| update\_limit\_size | Limit used during all updates. |
| update\_primary\_key\_columns | ‘True’ or ‘False’ to update primary keys. |
| where\_clasue\_early\_terminiation\_chance | When generating select queries this chance will be used to dictate how many where clauses make up a query. Where clauses will continue to get stacked onto each other until this chance is hit. 1 will results in single where clause queries, 100 will results in many long queries. |
| where\_clasue\_range\_chance | Chance a where clause will include a range. |