**Deep Data Bench Benchmark Analysis Walkthrough**

Adam Tetelman, 1/19/2016

**Deep Data Bench Summary:**

**What Is Deep Data Bench Used For?**

Deep Data Bench (DDB) is used to collect metadata describing a databases schema, cardinality, CRUD profile, and content. DDB can then use that metadata to re-create a synthetic dataset that will closely match the look and feel of the original. The synthetic dataset (hereafter referred to as data) can be generated to match the original size, be smaller, or be larger; all while retaining the original row count/cardinality ratios.

Some uses of DDB may call only for benchmarking the generation of data (Ingest only tests), but most will call for further analysis. After generating the original dataset, DDB allows a user to run diverse and highly tuned workloads against the data. DDB can model data growth, user growth, analytics, updates, ingest, transactional work, and any combination of those.

DDB allows you to run benchmarks either in separate phases (load, crud, analytics), or all at once (FLAQ – simultaneous inserts and analytics).

DDB allows a wide range of analytical queries to be generated using dials that control the where clause size, transaction size, for key relations, range sizes, and chance of an order by or group by clause.

**What Is Deep Data Bench Not Used For?**

While DDB allows for highly tuned data definition and diverse CRUD profiles, it is not ideal (currently) for generating high complexity Analytical queries. It is possible to generate a large dataset and run an existing complex query against it. However even this will require some massaging of the query being run or the values of the data generated. Most of the analytical queries generated by DDB are more in-line with an OLTP workload.

DDB is also not ideal for short duration tests. Anything that runs for less than 60 seconds may not have enough time to saturate the system and return realistic results.

**What Pieces make up a Deep Data Bench Analysis?**

A DDB Analysis is made up of 4 core pieces. The first will be the actual DDB tooling files.

The second will be a collection of metadata files that describe the data to generate and the CRUD profile to use. These are initially collected from a source database and then hand-tuned.

The third will be a collection of test configuration files that describe the connection information, which test profiles to run, the number of clients to use, and the test duration.

The fourth and final piece will be a collection of DDB Reports that are generated as results of a completed benchmark.

**Pre-Benchmarking:**

**Basic Problem Analysis:**

Before beginning any benchmark, you will want to run through the gambit of pre-benchmarking questions. Understand the use case. Understand the data size. Understand the growth potential. Understand the technologies being used. Understand the server size. Understand any in-place monitoring or other systems that the product being analyzed might connect to.

Be sure to read the DDB Documentation and validate that the problem being approached fits this tool.

**Environment Setup:**

Ensure that your test environment is setup with the Deep Data Bench required packages (python 2.7 and the MySQLdb Python 2.7 module).

**Benchmark Test Machines Using Other Tools:**

In many cases it is a good idea to do a quick benchmark of the test server using an external tool. This step is optional, but it is often helpful to have an understanding on how a server runs a basic IO test before running a DB benchmark.

Sysbench is a good benchmark to use for this. There is a lot of Sysbench documentation online and the tool can be used to do a quick and dirty CPU, IO, and Network benchmark before beginning the real heavy lifting of DDB. It is also possible to run a Sysbench MySQL test and compare that against DDB running the Sysbench profile with the Sysbench schema. These numbers should be almost identical. Note that the Sysbench IO test below is just one of many ways to run it.

*#Run a Baisc dd test*

*#This will give you a write speed such as 1.2 GB/s*

*#This should take ~3 minutes*

dd if=/dev/zero of=test1 bs=1M count=10240

*#Run a Basic Sysbench test*

*#This will give you a basic IO speed like 12.944Mb/sec*

*#This should take ~3 minutes*

*sudo apt-get install* -y *sysbench*

sysbench --test=fileio --file-total-size=10G prepare

sysbench --test=fileio --file-total-size=10G --file-test-mode=rndrw --init-rng=on --max-time=300 --max-requests=0 run

#Run a Sysbench (note this assumes MySQL is configured)

#This should return a transactions per second like 754.75 per sec.

#This should take ~3 minutes

mysql -uroot -pfoobar -e"create database sysbench; create database ddb;"

*sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=sysbench --mysql-user=root --mysql-password=foobar prepare*

*sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=sysbench --mysql-user=root --mysql-password=foobar --max-time=60 --oltp-read-only=on --max-requests=0 --num-threads=8 run*

*#Run a simulated Sysbench test with DDB*

*#This should give you a load time and a sysbench time*

*#The results from this test should match the previous sysbench test*

*#Note that the number you want to compare is the trx/second (21) \* avg\_trx\_size (42.21)*

bigdatabench.py --pillars="PureLoad,sysbench" --pillar\_durations="1000000,60s" --num\_clients="8,8" --pre\_generate\_data=False --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

#cleanup

mysql -uroot -pfoobar -e"drop database sysbench; drop database ddb;"

**Collecting Metadata:**

In order to run DDB, you will need to collect initial metadata from the source database. The ideal way to do this is to run the main DDB script somewhere that has full access to both the source and destination database. If this is not possible, the second best path is to run the metadata.py script somewhere that has access to the source database. It is also optimal to provide either the DDB or metadata script access to a slow query log during those steps. (If that is not possible it that data can be added manually later.) If neither of those two cases are possible, the third method of generating metadata requires a full schema dump from the source database and using that to create a new “source database” somewhere you have access to.

**Collecting Data Using the Metadata Script (second best):**

Use the following commands to generate a metadata file and save it off somewhere you can reference later. Note that what is captured from a slow query log may not actually be useful or fully representative. It is recommend *against* using that feature unless you have a full understanding of the use case and a large slow query log.

By default the metadata script will collect all the mins, maxes, and cardinality information from the source database. This can be disabled using the collect\_stats flag, however that is not recommended.

#This will Generate a full metadata object with schema, cardinality, etc.

#This should be rather quick, less than a minute

metadata.py --database=sysbench --mysql\_host=127.0.0.1 --mysql\_user=root --mysql\_password=foobar | tee /tmp/metadata.txt

#This will do the same but will also pull the CRUD ratios from the slow query log for each table.

metadata.py --database=sysbench --mysql\_host=127.0.0.1 --mysql\_user=root --mysql\_password=foobar --slow\_query\_file=/var/lib/mysql/slow.log

**Collecting Data Using the Metadata Script Without Access to the Source (third best):**

Collecting metadata without access to the source database works the same way that it does when you have access to the source database. The difference here is that you must first have access to a full schema dump. You must take your full schema dump and load it into a local database. After doing that simply run the above commands.

This will generate a valid metadata object. However, it will be missing the cardinality and min/max information. This should be identical to a metadata object collected from the source data with the collect\_stats flag disabled.

Note that if you are not doing any further tuning of your metadata file this “new source database” could actually be polled directly by the main DDB script.

**Adding Parsed Slow Query Log to Existing Metadata File (optional):**

If you gain access to the slow query logs after generating the metadata, then the data can be manually added by running the ParseSlowQuery.py script and hand editing the metadata file.

Note that if there are any tables not referenced in the slow query logs you will have to manually add an entry for them or you may have unexpected results.

In order to do this hand edit you will need to take the output from the Slow Query Parser and manually copy the values for each CRUD type to the appropriate table section. See the below metadata object and Parsed Slow Query example. Note at the time of writing this is future work to improve this process.

ParseSlowQuery.py --file=/var/lib/mysql/slow.log | tee parsed-slow.log

#This is the output generated from the Slow Query Parser

{

"DELETE": {

"all\_tables": 840,

"sbtest": 840

},

"INSERT": {

"all\_tables": 3341,

"sbtest": 3341

},

"REPLACE": {

"all\_tables": 0

},

"SELECT": {

"all\_tables": 48467,

"information\_schema.columns": 14,

"information\_schema.statistics": 22,

"information\_schema.tables": 23,

"sbtest": 48408

},

"UPDATE": {

"all\_tables": 2523,

"sbtest": 2523

}

}

#This is the matching section from the metadata object

{

"sbtest": {

"CRUD": {

"DELETE": 840,

"INSERT": 3341,

"REPLACE": 0,

"SELECT": 48404,

"UPDATE": 2523

},

"SIZE": 55108,

"SLEEP": {

"MAX": 0,

"MIN": 0

},

"TABLE\_TYPE": "BASE TABLE"

}

},

**Use Deep Data Bench to Collect Metadata File (optimal):**

This is the optimal method and the most simple. In fact, if you are simply running a standard set of DDB profiles without any hand tuning (or you have added new profiles to the metadata.py script), you can skip this step altogether and DDB will generate the metadata at run time.

If you plan to hand tune the metadata files at all, you will essentially want to run a very short duration of DDB. This will poll out to the source database, generate a metadata file, and then do a short DDB test on your destination server. When this test is complete, you can drop the database that was created and save off the metadata object from the meta\_data folder.

#This will run a short DDB

#Because it is less than 1000 rows duration no data will be generated

#The metadata object will be stored in meta\_data/<profile-name>.json

#In this case save off /meta\_data/PureLoad.json for hand editting

bigdatabench.py --pillars="PureLoad " --pillar\_durations="100" --num\_clients="1" --pre\_generate\_data=False --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

cp ./meta\_data/PureLoad.json ./meta\_data/demo-sysbench.json

**Validating Metadata File:**

At this point in the process you should have generated a single metadata object that fully represents your environment. Save this off named something special; later in the process you will be making copies of this file and tuning each one separately.

It is recommended that you do a quick scan of the file, validate that all the min/max/uniqueness values make sense, and validate the CRUD ratios are sane. (You will most likely be modifying these later during load time.)

**Locating Interesting Queries:**

This is also a good time in the process to identify any interesting/complex queries that are an ideal test case. These can be identified either by expert knowledge or by running a quick analysis on the slow query log using the Percona toolkit. You should save these for testing later.

#This will generate a slow query digest that outlines the top 10 queries

#It will let you know how often and how long all the queries are being run

#For large slow queries this can take a long time to run (15 minutes)

wget 'https://www.percona.com/downloads/percona-toolkit/2.2.16/deb/percona-toolkit\_2.2.16-1.tar.gz'

tar -xvf percona-toolkit\_2.2.16-1.tar.gz && cd percona-toolkit-2.2.16/bin

./pt-query-digest /var/lib/mysql/slow.log | tee slow.log.digest

**Designing a Proper Test:**

**Determining if the Collected Data is Valid:**

After collecting the metadata and validating it is correct, it is worth taking a few extra minutes to validate that the database you are benchmarking is correct, the tables are correct, and the use case is still being replicated.

At this point, you will also want to give further consideration to the actual workload being simulated and the CRUD ratios to be checked. If no valid slow query log was provided ensure you have a basic use case to test such as FLAQ, HTAP, Heavy Ingest, or “slight inserts with heavy updates and heavy reads, no deletes”. You will need an understanding of the use case in the next sections.

**Determining if a Custom Profile is Necessary:**

If you are running the main DDB script, there are many cases where you do not actually need to pre-generate or tune a metadata file. However, it will always be necessary if you are manually running the metadata script. In which case the “profiles” that you run are synonymous with the metadata files you create and built in profiles cannot easily be used.

DDB has several built in profiles that do a fairly good job at simulating many workloads. Take a quick look at the table describing these in the DDB documentation. If the workload being simulated only requires ingest, even amounts of CRUD, or analytics it is probable that you will not actually need to generate a custom metadata object.

If you are using the main DDB script and your testing interests are covered in that table, you can skip some of the tuning steps and come back to them if you are having issues with your completed benchmark.

**Determine Possible Growth Factors:**

Before designing the test you will want to have an understanding of future scenarios. The power behind DDB is that it allows you to benchmark your data today and your data tomorrow, with the same test.

Think a little bit about how large your data set may be in a year and in 5 years. Think of how many additional clients or worker threads you will have in 1 year and in 5 years. These are things we can easily model and plot against each other later.

**Determine Optimal Test Phases:**

DDB runs benchmarks in what are called “pillars”. These can be thought of as separate test phases. Each pillar is made up of 1 or more profiles, a duration to run those profiles for, and the number of clients to run each of those profiles.

A typical DDB may have pillars=”PureLoad,EvenCRUD,Analytics”, which will run a large load of the database, run a transactional workload, and follow up with some Analytics.

Another typical use case may be to run DDB twice. Once with pillars=”PureLoad” to generate a very large dataset. Saving off that dataset for future use, and then running another DDB test with pillars=”JustInserts Anayltics” and retain\_destination\_database=True. This second test would continue to do inserts on the dataset while simultaneously doing analytical queries. It will also retain all the previously loaded data.

Determine whether or not you want your initial load to be a part of the benchmark. If you are want to run several different benchmarks and are dealing with 100s of billions of rows you may not want to continuously reload that data if Ingest isn’t your primary concern.

Determine which phases you are interested in after that. You may be interested in pure analytical performance and pure transactional performance. You may also be interested in modeling future or current system behavior by doing many different tasks simultaneously. Come up with a plan for this and for each pillar that you would like that is not a predefined profile you will need to hand-craft the metadata later.

**Identify Potential Schema Improvements:**

There is a lot to identifying potential schema improvements and most of that is unrelated to DDB. This section is a brief overview.

When doing benchmarking there are a few basic things to look out for: indexes that can be added or removed, query that can be broken down or re-written, transactions that are too large or too small, data types that are unnecessarily large, and so on. When you have identified one of these schema changes using DDB, validating the change is fairly straightforward.

Continue on with setting the original benchmark up. Once that is completed and working copy all the metadata files you used to generate that benchmark and make the schema modifications you would like to test in those files. After you have done that, simply re-run the test using the modified files.

Similarly if you are not generating metadata files and are simply running the main DDB script, you can just modify the schema on your source database and re-run the benchmark.

The results of the benchmark should give you a good idea how your schema changes affected the grand scheme of things. If things are the same or better, go ahead and run any specific queries to be accelerated and verify the results. When doing testing like this it is also important to keep an eye on overall disk size; adding indexes can increase the disk footprint.

**Generating Initial Data Set:**

**Initial Metadata Tuning:**

If you are running your benchmark using the main DDB script and have not manually generated a metadata file using metadata.py you may skip this step.

Before generating an initial dataset, you will need to tune your metadata file for a fast load. This can be done by copying your metadata file (name it PureLoad-mydatabase.json), and modifying a few sections by hand.

In the CRUD section, you will want to make sure all DELETE, REPLACE, SELECT, UPDATE, MIIN, MAX values are 0. Make sure INSERT is 1000.

In the settings section, you will want to set extended\_insert\_size to 400.

If there are any changes to max values/min values, make them now in the top section of your metadata file.

If there are is any fixed\_data you would like for a specific table.column, add that to the bottom section now.

At the end of this step, you will have created a PureLoad-mydatabase.json file that will quickly and accurately load your data.

#Sample min/max section, note current values are null

"sbtest": [

{

"column\_name": "id",

"current\_max": null,

"current\_min": null,

"datatype": "int",

"foreign\_keys": [],

"max": 10000000000,

"method": "autoinc",

"min": 12345,

"uniqueness": 100

#Sample CRUD section, note only INSERT is non-zero

"sbtest": {

"CRUD": {

"DELETE": 0,

"INSERT": 1000,

"REPLACE": 0,

"SELECT": 0,

"UPDATE": 0

},

"SIZE": 1000109,

"SLEEP": {

"MAX": 0,

"MIN": 0

#Options section, note as this is insert only most of these options do nothing.

"commit\_early\_chance": 0,

"create\_savepoint\_chance": 0,

"delete\_limit\_size": 10,

"extended\_insert\_size": 400,

"max\_tables\_in\_a\_transaction": 30,

"max\_transaction\_size": 1,

"min\_transaction\_size": 1,

"pre\_generate\_data": false,

"range\_max\_date\_days": 1,

"range\_max\_float": 1000,

"range\_max\_int": 1000,

"range\_max\_time\_minutes": 4,

"range\_max\_year": 1,

"release\_savepoint\_chance": 0,

"rollback\_chance": 0,

"rollback\_early\_chance": 0,

"rollback\_to\_savepoint\_chance": 0,

"scale\_factor": 1,

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

"select\_join\_chance": 2,

"select\_limit\_size": 1000,

"select\_lock\_in\_share\_mode\_chance": 0,

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

"update\_limit\_size": 10,

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

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

"where\_clause\_range\_chance": 0

#Sample fixed data section table sbtest column c will have a 1/3 chance of being option-1 a 1/3 chance of being option-2 and a 1/3 chance in being random

"sbtest.c": [

"option-1",

"option-2",

"\*"

]

}

**Running an Insert Only Workload:**

If you have opted to do an initial load only benchmark, or are purely interested in ingest, the process is going to be similar to running other DDB tests. The one exception where PureLoad scenarios differ from other benchmarks is that you will want to ensure that the pre\_generate\_data option is set to False. Having this set to true when loading very large data sets can cause DDB to blow out of memory or run very slowly.

Earlier we generated the demo-sysbench.json file using the PureLoad profile. We then further tuned this file and will use it to run an initial load in the sample below. Note that if we had not originally used the PureLoad profile to generate that file we would need to tune the CRUD ratios (0 for everything but INSERT), and the query options (extended\_insert\_size=400, pre\_generate\_data=False).

Depending on how you would like to run benchmarks following the load you have two options. You can either backup the generated dataset and delete/reset the dataset you are working with after each run, or you can simply continue to grow and manipulate the same working dataset with each benchmark. Which method you use is highly dependent on your data size and what is most of interest. However creating a backup, running a benchmark, deleting the active database, and copying over the backup will always results in a more accurate A/B assessment.

Regardless of which of those two methods you use it is important that all future DDB runs use the retain\_destination\_database flag or you data will be lost.

#This will run 32 threads doing extended inserts of size 400.

#It will continue until a total of 100 billion rows have been loaded

#Both methods will result in identical workloads

#Running using the handmade profile, note there is no source information

bigdatabench.py --pillars="./meta\_data/demo-sysbench.json" --pillar\_durations="100000000000" --num\_clients="32" --pre\_generate\_data=False --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

#Running using a default profile with access to the source database

bigdatabench.py –pillars=”PureLoad" --pillar\_durations="100000000000" --num\_clients="32" --pre\_generate\_data=False --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

**Running Standard Ingest, Transactional, and Analytical Tests:**

**Where to run the test:**

While it is possible to run DDB on the same server as your test database, it is highly recommended that DDB has a dedicated server with enough cores and disk space to drive a load.

**Configure MySQL:**

Before running any benchmarks, you should make sure that MySQL has been properly configured for the use case you are testing. It is also a wise step to restart MySQL in between every benchmark.

**How to save test configurations:**

For repeatable and complex testing, DDB allows you to specify test parameters in at the command line or via a config file (see the documentation). The examples in this document all use the command line method, for ease of repeatability it is recommended that you use a config file for each test and run DDB with the config file you would like to test.

bigdatabench.py --conif load-test.ini

**Running a Time Based Test Versus a Transaction Based Test:**

One of the first decisions you will come across with a standard benchmark is whether to run the test for a set time period or for a certain number of transaction. With DDB you can run multiple pillars that use one method or the other.

It is typically recommended that during load periods a benchmark specify a number of rows for the duration.

For stress testing, it is recommended to use a duration.

**Determining how many cores to use:**

When running a DDB benchmark, you will typically want the total num\_clients to be equal to the number of cores on the machine.

pillars=”PureLoad, EvenCRUD”, num\_clients= “32, 32”

If a pillar is made up of two or more profiles this means they should sum to the number of cores on the system.

pillars=”PureLoad, EvenCRUD Random Analytics”, num\_clients= “32,16 8 8”

**Tuning a metadata object to match a profile:**

If you are forced to use a custom metadata object and you would like to mimic the default DDB profiles, you will need to follow the following steps.

Copy the metadata file to <profile>-<database>.json (for this example I will use TRX\_CRUD). There will now be two sections in your metadata file you must edit: the CRUD ratio section and the options section. In order to properly mimic the DDB profiles you will have to look inside the metadata.py file, locate the settings that would be used, and make the appropriate changes in your metadata file.

#Located in the \_\_init\_\_ function

#You will want to copy set these values in the options section of your metadata file

#You will also want to ensure that the settings not mentioned under the profile you are matching match the defaults set at the top of the \_\_init\_\_ function

elif self.profile == "TRX\_CRUD":

self.global\_options['pre\_generate\_data'] = True

self.global\_options['extended\_insert\_size'] = 5

self.global\_options['min\_transaction\_size'] = 1

self.global\_options['max\_transaction\_size'] = 200

self.global\_options['rollback\_chance'] = 10

self.global\_options['select\_lock\_in\_share\_mode\_chance'] = 20

#Located in the \_\_get\_what\_to\_do function

#Copy these values into the appropriate CRUD section in your metadata.

#You will need to update all CRUD values for every table

elif self.profile == "TRX\_CRUD":

return { 'CRUD' : {'INSERT' : 100, 'SELECT' : 100, 'UPDATE' : 100, 'DELETE' : 100, 'REPLACE' : 100}, 'SLEEP' : {'MIN' : 0, 'MAX' : 0} }

**Tuning for Proper Analytics:**

For a complete list of all the tuning options available in DDB see the DDB documentation.

To properly setup analytics in a DDB, there are a few things you must do.

Before starting the test, you should have properly added fixed data options to your metadata. You should also have added any foreign key relationships. During load, DDB will honor any FK restraints listed in the metadata file. If your source database originally had MySQL foreign key constraints, these should have automatically been added.

If your data is set up to look properly, you will want to modify all the “range”, “select” and “where” options in the DDB metadata file.

The range values define how big a range to query against.

The where options dictate how long the where clause will be, they also define how often to do range queries versus point queries.

The select options dictate the chance of joins, order bys, group bys, and limit the size of any single response.

Modify these options to roughly match your products use case. If you are unsure what to set these to match them to the Analytics profile.

**Running a Basic 3-Pillar Test:**

The basic 3-pillar test is made up of a load pillar, a transactional pillar, and an analytics pillar. This can be run with the following commands.

bigdatabench.py –pillars=”PureLoad,EvenCRUD,Analytics" --pillar\_durations="1000000000,1000000,100000" --num\_clients="32,32,32" --pre\_generate\_data=True --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

**Running a Basic FLAQ Test:**

To test FLAQ you will want to do a heavy load followed by a period of load and analytics. This can be accomplished with the below command.

bigdatabench.py –pillars=”PureLoad,Analytics JustInserts" --pillar\_durations="1000000000,1000s" --num\_clients="32,8 24" --pre\_generate\_data=True --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

**Running a Basic Ingest Test:**

A basic ingest test will be the same as the initial data load mentioned above. If your production environment does not use extended inserts, use the JustInserts profile instead of PureLoad.

bigdatabench.py –pillars=”PureLoad " --pillar\_durations="1000000000 " --num\_clients="32” --pre\_generate\_data=True --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

**Running Custom Tests:**

**How to Modify Metadata Dials:**

As mentioned above, there are many dials of control with DDB. If you feel as though the basic profiles do not cover your use case, you can create a copy of your metadata file and manually tweak any values. Typical custom tests will tweak the CRUD ratios and the corresponding DDB option settings.

Rather generating a metadata file and then tweaking it, it is also possible to add a profile by editing the metadata.py file. In fact, if you are able to run DDB somewhere with access to the destination and the source database, this is preferable. To modify the script simply add a CRUD section and an options setting in the metadata.py script and then either run metadata.py with your profile or run DDB.

**How to Simulate Complex Workloads in a Test Config:**

For complete details on the options see the DDB documentation.

As previously mentioned, all the command line options can be put into a configuration file. There is no limit to the number of pillars a test can have. There is no limit to the number of profiles that can make up a pillar.

DDB also has a repeate\_x\_times flag so you can write a test in such a way that it create a pattern, running an ingest, then some analytics, then repeating the cycle. This can be a good way to run the same test over and over again on large datasets each time.

**Running an A/B Benchmark:**

**Comparing MySQL Engines:**

Comparing MySQL Engines with DDB is simple. Generate all your metadata files. Create all your test configuration files. The only change that you will need to make is to make a copy of all your test files and add a destination\_mysql\_engine field to each of them. If you were comparing InnoDB vs DeepSQL and running PureLoad, EvenCRUD, Analytics you would end up with the following files:

test\_configs: myload.deep.ini, mytest.deep.ini, myload.innodb.ini, mytest.innodb.ini

meta\_data: PureLoad.json, EvenCRUD.json, Analytics.json

After doing that, simply run the test as you would with the first set of files, followed by the second set of files. Be sure to copy out the reports from each iteration from the reports directory. And be sure not to run out of disk space while doing this testing.

As an optional step, you may also want to store the data for each engine in a separate database so you can play around with them at the end of testing.

It is also very important that when A/B testing engines you restart MySQL between each run and verify your database is properly tuned for the engine you are running.

**Comparing MySQL Configurations:**

Comparing MySQL configuration changes with DDB is a little bit harder. The interaction that DDB has with the MySQL configurations is that it stores the output of ‘show variables’ in the report files.

If you would like to A/B test MySQL configuration changes, it is recommended that you create several different my.ini files with the changes you would like to test. You should then write a script that is responsible for restarting MySQL with the proper my.ini file, clearing system cache, and running DDB with the report\_name flag used. This flag will allow you to differentiate between runs.

**Comparing Schema Changes:**

This can be in three ways. You can make the schema change in your source database and allow DDB to pick up the change. Alternatively, you can modify your destination database with an alter command and then re-run your test with the retain\_destination\_database flag. Note that this will only work if you followed previous directions to break your test into an initial load command and a second test phase. This will also only work if you are modifying indexes, not if you are adding columns.

The last method is to make a copy of all your metadata files and update the schema information in those files. The schema is kept in plain text in the metadata and you can simply edit the schema and re-run your test. Note that any data would need to be regenerated. Note that you must also edit the indexes section of the metadata files if you are adding a new index.

#Sample Indexes and Schema section of Metadata file

{

"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 AUTO\_INCREMENT=1000001 DEFAULT CHARSET=latin1"

},

**Simulating Data Growth and User Growth Over Time:**

Often times after running your benchmark, you will want to re-run the same benchmark at higher scale. You should have determined possible growth patterns in a previous step, the below sections go over some samples on how to test scale.

DDB is very versatile and these test can be run many ways. They can be done in separate test runs, using the repeate\_x\_times flag, or done all in the same DDB test. The samples below will utilize a single benchmark approach.

Note that the pre\_generate\_data flag is useful in these cases.

**Watching Data Size Grow During a Single Test:**

The below test will run an identical 100,00 transaction OLTP workload on a 1M, 10M, and 100M row dataset.

bigdatabench.py –pillars=”PureLoad,EvenCRUD,PureLoad,EvenCRUD, PureLoad,EvenCRUD" --pillar\_durations="1000000,100000,9000000,100000,90000000,100000" --num\_clients="32,32,32,32,32,32" --pre\_generate\_data=False --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

**Watching Thread Count Grow During a Single Test:**

The below will run a total 100,000 medium sized OLTP transaction each with 1,2,4,8,16, and 32 simultaneous clients.

bigdatabench.py –pillars=”TRX\_CRUD,TRX\_CRUD,TRX\_CRUD,TRX\_CRUD,TRX\_CRUD,TRX\_CRUD " --pillar\_durations="100000, 100000, 100000, 100000, 100000, 100000" --num\_clients="1,2,4,8,16,32" --pre\_generate\_data=False --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

**Replicating a Single Test at Multiple Scale Factors:**

If you would like to exactly scale up a dataset while maintaining table size ratios that match the source database, you can also modify the scale\_factor flag in the metadata file. The default value for this flag is 1 and it is used to dictate the maximum values that can be selected for a column.

If your metadata file has maximums set, running with more and more rows would eventually lead to duplicates and a maximum database size. If your id has a max value of 65000, you can only have 65000 unique rows; if you set the scale\_factor value 2 to the new max will become 130000.

If you are using this method you will need to create several metadata file each with a different scale factor. A sample command would look like this:

bigdatabench.py –pillars=”./meta\_data/PureLoad-sf1.json,EvenCRUD,./meta\_data/PureLoad-sf10.json,EvenCRUD, ./meta\_data/PureLoad-sf100.json,EvenCRUD" --pillar\_durations="1000000,100000,9000000,100000,90000000,100000" --num\_clients="32,32,32,32,32,32" --pre\_generate\_data=False --source\_mysql\_user=root --source\_mysql\_password=foobar --source\_mysql\_host=127.0.0.1 --source\_database=sysbench --destination\_mysql\_user=root --destination\_mysql\_password=foobar --destination\_mysql\_host=127.0.0.1 --destination\_database=ddb

**Analyzing Deep Data Bench Reports:**

**Using the Report Viewer:**

The DDB report viewer allows you to review results from a benchmark, compare them to other benchmarks, view all the MySQL errors and warning, and will display the slowest queries it saw along with their explain output.

Assuming you have either saved off all your benchmark reports or are leaving them in the default reports directory, you simply point the tool at the file and specify the flags for what you would like to view.

Note that seeing several duplicate key errors in the report is to be expected. This is due to the random nature of inserts.

ReportViewer.py --help

usage: ReportViewer.py [-h] [--summary] [--json\_summary] [--vars] [--db\_info]

[--errors] [--warnings] [--slowest\_queries] [--trend]

ReportViewer.py --summary --errors --warnings reports/sysbench\_reportGxHUxy.dump

3 Pillar Bench Report

------------------------------------------------------------------------------------------

Pillar: PureLoad

number of clients: 8

average client duration: 13.1866 (sec)

test duration: 13.6544 (sec)

Query Execution Time (seconds)

Operation Count Rate (per sec) rows\_affected Min Max Total Average

INSERT 1,001,600 73,353.64 1,001,599 0.0095 0.1155 104.0350 0.0001

Num Transactions: 2504

Ave Transaction Size: 1

Transactions Per Second: 183.384110616

------------------------------------------------------------------------------------------

Pillar: sysbench

number of clients: 8

average client duration: 60.0197 (sec)

test duration: 60.0505 (sec)

Query Execution Time (seconds)

Operation Count Rate (per sec) rows\_affected Min Max Total Average

INSERT 587 9.78 204 0.0042 0.0166 1.6710 0.0028

BEGIN 2,401 39.98 0 0.0038 0.0098 13.5008 0.0056

UPDATE 1,675 27.89 1,675 0.0038 0.0331 13.2409 0.0079

COMMIT 2,393 39.85 0 0.0038 0.0335 17.2209 0.0072

SELECT 47,496 790.93 33,066 0.0037 0.0656 383.5844 0.0081

DELETE 822 13.69 822 0.0039 0.0328 6.2124 0.0076

Num Transactions: 2393

Ave Transaction Size: 20

Transactions Per Second: 39.8497930908

Profile: PureLoad

Profile: sysbench

QueryGenerator-9: 1062 - Duplicate entry '939151' for key 'PRIMARY' --count---> 169

QueryGenerator-10: 1062 - Duplicate entry '947068' for key 'PRIMARY' --count---> 161

QueryGenerator-11: 1062 - Duplicate entry '2' for key 'PRIMARY' --count---> 177

QueryGenerator-12: 1062 - Duplicate entry '992168' for key 'PRIMARY' --count---> 168

QueryGenerator-13: 1062 - Duplicate entry '4' for key 'PRIMARY' --count---> 174

QueryGenerator-14: 1062 - Duplicate entry '122704' for key 'PRIMARY' --count---> 163

QueryGenerator-15: 1062 - Duplicate entry '711969' for key 'PRIMARY' --count---> 187

QueryGenerator-16: 1062 - Duplicate entry '356925' for key 'PRIMARY' --count---> 176

Profile: PureLoad

Profile: sysbench

**Comparing Multiple Reports:**

If you have tested multiple MySQL engines, configurations, or schemas the report viewer makes it easy to compare those results.

Simply pass the files you wish to compare to the report viewer as a comma separated list and it will do the comparisons for you.

Note that this allows you to compare any number of report files.

ReportViewer.py --summary reports/sysbench\_reportA0twBO.dump,reports/sysbench\_reportvJc36S.dump

............................................................................

PureLoad (INSERT) | sysbench\_reportA0twBO | sysbench\_reportvJc36S |

............................................................................

Count | 1,001,600 | 1,001,600 |

Rate (per sec) | 84,150 | 71,228 |

rows\_affected | 1001599.0000 | 1001599.0000 |

Min | 0.0086 | 0.0082 |

Max | 0.0965 | 0.1924 |

Total | 92.9071 | 107.4499 |

Average | 0.0001 | 0.0001 |

............................................................................

sysbench (INSERT) | sysbench\_reportA0twBO | sysbench\_reportvJc36S |

............................................................................

Count | 3,087 | 3,042 |

Rate (per sec) | 10 | 10 |

rows\_affected | 1155.0000 | 1087.0000 |

Min | 0.0040 | 0.0038 |

Max | 0.0328 | 0.0168 |

Total | 8.9709 | 9.5376 |

Average | 0.0029 | 0.0031 |

............................................................................

sysbench (BEGIN) | sysbench\_reportA0twBO | sysbench\_reportvJc36S |

............................................................................

Count | 11,495 | 11,460 |

Rate (per sec) | 38 | 38 |

rows\_affected | 0.0000 | 0.0000 |

Min | 0.0038 | 0.0037 |

Max | 0.0102 | 0.0171 |

Total | 69.2545 | 70.5616 |

Average | 0.0060 | 0.0062 |

............................................................................

sysbench (UPDATE) | sysbench\_reportA0twBO | sysbench\_reportvJc36S |

............................................................................

Count | 7,937 | 7,929 |

Rate (per sec) | 26 | 26 |

rows\_affected | 7939.0000 | 7934.0000 |

Min | 0.0038 | 0.0038 |

Max | 0.1153 | 0.0335 |

Total | 65.8935 | 66.1621 |

Average | 0.0083 | 0.0083 |

............................................................................

sysbench (COMMIT) | sysbench\_reportA0twBO | sysbench\_reportvJc36S |

............................................................................

Count | 11,487 | 11,452 |

Rate (per sec) | 38 | 38 |

rows\_affected | 0.0000 | 0.0000 |

Min | 0.0037 | 0.0037 |

Max | 0.0336 | 0.0649 |

Total | 86.1058 | 85.9953 |

Average | 0.0075 | 0.0075 |

............................................................................

sysbench (SELECT) | sysbench\_reportA0twBO | sysbench\_reportvJc36S |

............................................................................

Count | 227,909 | 227,430 |

Rate (per sec) | 760 | 758 |

rows\_affected | 159684.0000 | 158102.0000 |

Min | 0.0037 | 0.0037 |

Max | 0.1158 | 0.1154 |

Total | 1916.3730 | 1918.3028 |

Average | 0.0084 | 0.0084 |

............................................................................

sysbench (DELETE) | sysbench\_reportA0twBO | sysbench\_reportvJc36S |

............................................................................

Count | 3,977 | 3,935 |

Rate (per sec) | 13 | 13 |

rows\_affected | 3979.0000 | 3935.0000 |

Min | 0.0037 | 0.0038 |

Max | 0.0331 | 0.0336 |

Total | 31.9140 | 31.9727 |

Average | 0.0080 | 0.0081 |

............................................................................

Pillar Scores:

............................................................................

| sysbench\_reportA0twBO | sysbench\_reportvJc36S |

............................................................................

PureLoad | 84,150 | 71,228 |

sysbench | 886 | 884 |

............................................................................

Total | 85,037 | 72,112 |

............................................................................

**Understanding the Margins of Error:**

Because these tests are random, it is important to note that each test run may be different and there is a margin of error. Typically in a healthy setup where the database and DDB server are separate, the pre\_generate\_data flag is used, and the test duration is longer than 15 minutes, the margin of error tends to be below 5%.

That being said if results look suspicious, attempt to re-run/duplicate the results first if possible.

**Analyzing Report Results:**

The reports give a transaction per second rate for each CRUD type as well as an average. When analyzing the results, be sure you are looking at all of the numbers and not just the average.

It is also important to verify there were no critical errors found in the report (such as a crashed MySQL).

**Summary of Deep Data Bench Usage:**

1. Determine if you can test where you have access to your source database
2. Determine if the default DDB profile can simulate your workloads
3. If the answer to those previous two questions was no generate a metadata object and create several tuned copies for your test
4. Run an initial load if you are working with very large data
5. Validate your MySQL configuration between each benchmark
6. Run your benchmarks
7. Scale your benchmarks
8. Analyze the results