

Basho Bench

Contents

1. [Download](#)
 2. [Documentation](#)
 3. [Installation](#)
 4. [Usage](#)
 5. [Generating Benchmark Graphs](#)
 6. [Configuration](#)
 7. [Custom Driver](#)
-

Basho Bench is a benchmarking tool created to conduct accurate and repeatable performance tests and stress tests, and produce performance graphs.

Originally developed by Dave Smith (Dizzy) to benchmark Riak, Basho's key/value datastore, it exposes a pluggable driver interface and has been extended to serve as a benchmarking tool against a variety of projects. New drivers can be written in Erlang and are generally less than 200 lines of code.

Download

The main repository for basho_bench is http://github.com/basho/basho_bench/ .

Documentation

Note on Documentation

This topic replaces the documentation that was in the basho_bench repository under docs/Documentation.org prior to February 2011.

How does it work?

When Basho Bench starts (`basho_bench.erl`), it reads the configuration (`basho_bench_config.erl`), creates a new results directory, then sets up the test. (`basho_bench_app.erl`/`basho_bench_sup.erl`)

During test setup, Basho Bench creates:

- One **stats process** (`basho_bench_stats.erl`). This receives notifications when an operation completes, plus the elapsed time of the operation, and stores it in a histogram. At regular intervals, the histograms are dumped to `summary.csv` as well as operation-specific latency CSVs (e.g. `put_latencies.csv` for the 'put' operation).
- N **workers**, where N is specified by the `concurrent` configuration setting. (`basho_bench_worker.erl`). The worker process wraps a driver module, specified by the `driver` configuration setting. The driver is randomly invoked using the distribution of operations as specified by the `operations` configuration setting. The rate at which the driver invokes operations is governed by the `mode` setting.

Once these processes have been created and initialized, Basho Bench sends a run command to all worker processes, causing them to begin the test. Each worker is initialized with a common seed value for random number generation to ensure that the generated workload is reproducible at a later date.


During the test, the workers repeatedly call `driver:run/4`, passing in the next operation to run, a keygen function, a valuegen function, and the last state of the driver. The worker process times the operation, and reports this to the stats process when the operation has completed.

Finally, once the test has been run for the duration specified in the config file, all workers and stats processes are terminated and the benchmark ends. The measured

latency and throughput of the test can be found in `./tests/current/`. Previous results are in timestamped directories of the form `./tests/YYYYMMDD-HHMMSS/`.

Installation

Prerequisites

- Erlang must be installed. See [Installing Erlang](#) for instructions and versioning requirements.
- [R statistics language](#)  must be installed if you wish to generate graphs (see the [Generating Benchmark Graphs](#) section, below).

Building from Source

Basho Bench is currently available as source code only. To get the latest code, clone the `basho_bench` repository:

Shell

```
$ git clone git://github.com/basho/basho_bench.git
$ cd basho_bench
$ make
```

Usage

Run `basho_bench`:

Shell

```
./basho_bench myconfig.config
```

This will generate results in `tests/current/`. You will need to create a

configuration file. The recommended approach is to start from a file in the `examples` directory and modify settings using the [Configuration](#) section below for reference.

Generating Benchmark Graphs

The output of `basho_bench` can be used to create graphs showing:

- Throughput - Operations per second over the duration of the test.
- Latency at 99th percentile, 99.9th percentile and max latency for the selected operations.
- Median latency, mean latency, and 95th percentile latency for the selected operations.

Prerequisites

The R statistics language is needed to generate graphs. Note: If necessary, R can be installed on a different machine than the one running `basho_bench`, and the performance data can be copied (via `rsync`, for example) from the load testing machine to the one that will be generating and viewing the graphs (such as a desktop).

- More information: <http://www.r-project.org/>
- Download R: <http://cran.r-project.org/mirrors.html>

Follow the instructions for your platform to install R.

Generating Graphs

To generate a benchmark graph against the current results, run:

Shell

```
make results
```

This will create a results file in `tests/current/summary.png`.

You can also run this manually:

Shell

```
priv/summary.r -i tests/current
```

Configuration

Basho Bench ships with a number of sample configuration files, available in the `/examples/` directory.

Global Config Settings

mode

The **mode** setting controls the rate at which workers invoke the `{driver:run/4}` function with a new operation. There are two possible values:

- `{max}` — generate as many ops per second as possible
- `{rate, N}` — generate N ops per second, with exponentially distributed interarrival times

Note that this setting is applied to each driver independently. For example, if `{rate, 5}` is used with 3 concurrent workers, `basho_bench` will be generating 15 (i.e. $5 * 3$) operations per second.

Erlang

```
% Run at max, i.e.: as quickly as possible
{mode, max}
```

```
% Run 15 operations per second per worker
```

```
{mode, {rate, 15}}
```

concurrent

The number of concurrent worker processes. The default is 3 worker processes. This determines the number of concurrent clients running requests on API under test.

Erlang

```
% Run 10 concurrent processes  
{concurrent, 10}
```

duration

The duration of the test, in minutes. The default is 5 minutes.

Erlang

```
% Run the test for one hour  
{duration, 60}
```

operations

The possible operations that the driver will run, plus their “weight” or likelihood of being run. Default is `[{get, 4}, {put, 4}, {delete, 1}]` which means that out of every 9 operations, 'get' will be called four times, 'put' will be called four times, and 'delete' will be called once, on average.

Erlang

```
% Run 80% gets, 20% puts  
{operations, [{get, 4}, {put, 1}]}.
```

Operations are defined on a **per-driver** basis. Not all drivers will implement the “get”/“put” operations discussed above. Consult the driver source to determine the valid operations. E.g., if you're testing the HTTP interface, the corresponding operations are “get” and “update” respectively.

If a driver does not support a specified operation (“asdfput” in this example) you may see errors like:

Log

```
DEBUG:Driver basho_bench_driver_null crashed: {function_clause,  
                                                [],  
                                                {{{basho_bench_worker,  
                                                  worker_next_op,1}}},  
                                                {{{basho_bench_worker,  
                                                  max_worker_run_loop,1}}}
```

driver

The module name of the driver that basho_bench will use to generate load. A driver may simply invoke code in-process (such as when measuring the performance of DETS) or may open network connections and generate load on a remote system (such as when testing a Riak server/cluster).

Available drivers include:

- `basho_bench_driver_http_raw` - Uses Riak's HTTP interface to get/update/insert data on a Riak server
- `basho_bench_driver_riakc_pb` - Uses Riak's Protocol Buffers interface to get/put/update/delete data on a Riak server
- `basho_bench_driver_riakclient` - Uses Riak's Distributed Erlang interface to get/put/update/delete data on a Riak server
- `basho_bench_driver_bitcask` - Directly invokes the Bitcask API
- `basho_bench_driver_dets` - Directly invokes the DETS API

On invocation of the `driver:run/4` method, the driver may return one of the following results:

- `{ok, NewState}` - operation completed successfully
- `{error, Reason, NewState}` - operation failed but the driver can continue processing (i.e. recoverable error)
- `{stop, Reason}` - operation failed; driver can't/won't continue processing
- `{'EXIT', Reason}` - operation failed; driver crashed

code_paths

Some drivers need additional Erlang code in order to run. Specify the paths to this code using the **code_paths** configuration setting.

key_generator

The generator function to use for creating keys. Generators are defined in `basho_bench_keygen.erl`. Available generators include:

- `{sequential_int, MaxKey}` - generates integers from 0..MaxKey in order and then stops the system. Note that each instance of this keygen is specific to a worker.
- `{partitioned_sequential_int, MaxKey}` - same as `{sequential_int}`, but splits the keyspace evenly among the worker processes. This is useful for pre-loading a large dataset.
- `{partitioned_sequential_int, StartKey, NumKeys}` - same as `partitioned_sequential_int`, but starting at the defined `StartKey` and going up to `StartKey + NumKeys`.
- `{uniform_int, MaxKey}` - selects an integer from uniform distribution of 0..MaxKey. I.e. all integers are equally probable.
- `{pareto_int, MaxKey}` - selects an integer from a Pareto distribution, such that 20% of the available keys get selected 80% of the time. Note that the current implementation of this generator MAY yield values larger than MaxKey due to the mathematical properties of the Pareto distribution.
- `{truncated_pareto_int, MaxKey}` - same as `{pareto_int}`, but will NOT yield values above MaxKey.
- `{function, Module, Function, Args}` - specifies an external function that should return a key generator function. The worker `Id` will be prepended to `Args` when the function is called.
- `{int_to_bin, Generator}` - takes any of the above `_int` generators and converts the number to a 32-bit binary. This is needed for some drivers that require a binary key.
- `{int_to_str, Generator}` - takes any of the above `_int` generators

and converts the number to a string. This is needed for some drivers that require a string key.

The default key generator is `{uniform_int, 100000}`.

Examples:

Erlang

```
% Use a randomly selected integer between 1 and 10,000
{key_generator, {uniform_int, 10000}}.

% Use a randomly selected integer between 1 and 10,000, as
{key_generator, {int_to_bin, {uniform_int, 10000}}}.

% Use a pareto distributed integer between 1 and 10,000; va
% will be returned 80% of the time.
{key_generator, {pareto_int, 10000}}.
```

value_generator

The generator function to use for creating values. Generators are defined in `basho_bench_valgen.erl`. Available generators include:

- `{fixed_bin, Size}` - generates a random binary of Size bytes. Every binary is the same size, but varies in content.
- `{exponential_bin, MinSize, Mean}` - generates a random binary which has an exponentially-distributed size. Most values will be approximately MinSize + Mean bytes in size, with a long-tail of larger values.
- `{uniform_bin, MinSize, MaxSize}` - generates a random binary which has an evenly-distributed size between MinSize and MaxSize.
- `{function, Module, Function, Args}` - specifies an external function that should return a value generator function. The worker `Id` will be prepended to `Args` when the function is called.

The default value generator is `{value_generator, {fixed_bin, 100}}`.

Examples:

Erlang

```
% Generate a fixed size random binary of 512 bytes
{value_generator, {fixed_bin, 512}}.

% Generate a random binary whose size is exponentially distributed
% starting at 1000 bytes and a mean of 2000 bytes
{value_generator, {exponential_bin, 1000, 2000}}.
```

rng_seed

The initial random seed to use. This is explicitly seeded, rather than seeded from the current time, so that a test can be run in a predictable, repeatable fashion.

Default is `{rng_seed, {42, 23, 12}}`.

Erlang

```
% Seed to {12, 34, 56}
{rng_seed, {12, 34, 56}}.
```

log_level

The **log_level** setting determines which messages Basho Bench will log to the console and to disk.

Default level is **debug**.

Valid levels are:

- debug
- info
- warn
- error

report_interval

How often, in seconds, should the stats process write histogram data to disk. Default is 10 seconds.

test_dir

The directory in which to write result data. The default is `tests/`.

basho_bench_driver_riakclient Settings

These configuration settings apply to the `basho_bench_driver_riakclient` driver.

riakclient_nodes

List of Riak nodes to use for testing.

Erlang

```
{riakclient_nodes, ['riak1@127.0.0.1', 'riak2@127.0.0.1']}
```

riakclient_cookie

The Erlang cookie to use to connect to Riak clients. Default is `'riak'`.

Erlang

```
{riakclient_cookie, riak}.
```

riakclient_mynode

The name of the local node. This is passed into [net_kernel:start/1](#).

Erlang

```
{riakclient_mynode, ['basho_bench@127.0.0.1', longnames]}.
```

riakclient_replies

This value is used for R-values during a get operation, and W-values during a put operation.

Erlang

```
% Expect 1 reply.  
{riakclient_replies, 1}.
```

riakclient_bucket

The Riak bucket to use for reading and writing values. Default is `<<"test">>`.

Erlang

```
% Use the "bench" bucket.  
{riakclient_bucket, <<"bench">>}.
```

basho_bench_driver_riakc_pb Settings

riakc_pb_ips

List of IP addresses to connect the workers to. A random IP will be chosen for each worker.

Default is `{riakc_pb_ips, [{127,0,0,1}]}`

Erlang

```
% Connect to a cluster of 3 machines  
{riakc_pb_ips, [{10,0,0,1},{10,0,0,2},{10,0,0,3}]}
```

riakc_pb_port

The port on which to connect to the PBC interface.

Default is `{riakc_pb_port, 8087}`

riakc_pb_bucket

The bucket to use for testing.

Default is `{riakc_pb_bucket, <<"test">>}`

basho_bench_driver_http_raw Settings

http_raw_ips

List of IP addresses to connect the workers to. Each worker makes requests to each IP in a round-robin fashion.

Default is `{http_raw_ips, ["127.0.0.1"]}`

Erlang

```
% Connect to a cluster of machines in the 10.x network
{http_raw_ips, ["10.0.0.1", "10.0.0.2", "10.0.0.3"]}.
```

http_raw_port

Select the default port to connect on for the HTTP server.

Default is `{http_raw_port, 8098}`.

Erlang

```
% Connect on port 8090
{http_raw_port, 8090}.
```

http_raw_path

Base path to use for accessing riak - usually `"/riak/"`

Defaults is `{http_raw_path, "/riak/test"}`.

Erlang

```
% Place test data in another_bucket
{http_raw_path, "/riak/another_bucket"}.
```

http_raw_params

Additional parameters to add to the end of the URL. This can be used to set riak r/w/dw/rw parameters as as desired.

Default is `{http_raw_params, ""}`.

Erlang

```
% Set R=1, W=1 for testing a system with n_val set to 1
{http_raw_params, "?r=1&w=1"}.
```

http_raw_disconnect_frequency

How often, in seconds or number of operations, the HTTP clients (workers) should forcibly disconnect from the server.

Default is `{http_raw_disconnect_frequency, infinity}`. (never forcibly disconnect)

Erlang

```
% Disconnect after 60 seconds
{http_raw_disconnect_frequency, 60}.

% Disconnect after 200 operations
{http_raw_disconnect_frequency, {ops, 200}}.
```

Custom Driver

A custom driver must expose the following callbacks.

Erlang

```
% Create the worker
% ID is an integer
new(ID) -> {ok, State} or {error, Reason}.

% Run an operation
run(Op, KeyGen, ValueGen, State) -> {ok, NewState} or {error, Reason}.
```

See the [existing drivers](#) for more details.

These May Also Interest You

- [Downloads](#)
- [Log Messages FAQs](#)

- Log Messages / FAQs
- Operating Riak FAQs
- Configuration Files
- Load Balancing and Proxy Configuration
- Configuring Secondary Indexes