**Performance Experiment - Data Stores**

The goal of this experiment is to benchmark how this technology performs when hardware resources are plentiful and the number of actors (publishers and subscribers) in the system varies.

Contact [@rich-java-dev](https://github.com/rich-java-dev) for clarifications.

**References for Workloads & Plugins**

[Netflix Data Benchmark (NDBench)](https://github.com/Netflix/ndbench/wiki) Netflix Data Benchmark (NDBench) is a pluggable cloud-enabled benchmarking tool that can be used across any data store system. NDBench provides plugin support for the major data store systems that we use -- Cassandra (Thrift and CQL), Dynomite (Redis), and Elasticsearch. It can also be extended to other client APIs.

[Yahoo! Cloud Serving Benchmark (YCSB)](https://github.com/brianfrankcooper/YCSB/wiki)

The goal of the YCSB project is to develop a framework and common set of workloads for evaluating the performance of different “key-value” and “cloud” serving stores. The project comprises two things:

The YCSB Client, an extensible workload generator The Core workloads, a set of workload scenarios to be executed by the generator Although the core workloads provide a well rounded picture of a system’s performance, the Client is extensible so that you can define new and different workloads to examine system aspects, or application scenarios, not adequately covered by the core workload. Similarly, the Client is extensible to support benchmarking different databases.

**Hardware Configuration**

Each test should be ran using Ubuntu 18.04 chameleon cloud VM’s with 8 CPU and 32 GB of RAM. If these hardware specs aren’t attainable, use the nearest available and document the configuration.

**Experiment Configurations**

**Raw Performance/Through-put**

The scope/purpose of this experiment is to help measure the read/write time averages across the different data stores, under various conditions.

Using the 5 samples provided (random char arrays of length 128, 254, 512, 1024, 10000 respectively) Here is the basic implementation of the random string for reference.

def randStr(chars = string.ascii\_uppercase + string.digits, N=10): ... return ''.join(random.choice(chars) for \_ in range(N))

10,000 iterations/data-points (delta times) collected for: -Reads -Writes (CREATE) for each payload of size in the provided samples: Using the 5 samples provided (random char arrays of length 128, 254, 512, 1024, 10000 respectively)

* run all writes over a range of keys 1-10000 (each payload being indexed by a unique/iterable)
* run all reads over the same given range (looking up each key once, for each record written.)
* run again with all writes/reads targeting the same key (eg: 1)

Collection the time-stamp before and after the Read/Write operation (as close to the actual Client API for the given data store)

Aim to run this experiment under the constraints for the following configurations:

| Reader | Writer | | 0 | 1 | | 1 | 0 | | 1 | 10 | | 10 | 1 | | 5 | 5 |

Provide output in the following format:

reader\_id, writer\_id, key, start\_time, end\_time, delta\_time eg: 0,1,12345.6789, 12345.6790, .0001

These are the configurations your program should be ran in.

| **Writer Machines** | **Writer Instances** | **Reader Machines** | **Reader Instances** |
| --- | --- | --- | --- |
| 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 10 | 10 | 1 | 1 |
| 1 | 1 | 10 | 10 |
| 5 | 25 | 5 | 25 |

If possible, implement these configurations with redundancy enabled and document the configuration.

These tests are described in detail below and have placeholders for results.

**Redundancy implementation Details**

Place the redundancy implementation details here.

**Preliminary Steps**

Calculate the following data using Excel or a custom program by analyzing the data in latencies.csv and throughput.csv

| **Metric** | **Value** |
| --- | --- |
| Write Latency Min |  |
| Write Latency Max |  |
| Write Latency Average |  |
| Write Latency Standard Deviation |  |
| Read Time Min |  |
| Read Time Max |  |
| Read Time Average |  |
| Read Time Standard Deviation |  |
| Write Throughput Min |  |
| Write Throughput Max |  |
| Write Throughput Average |  |
| Write Throughput Standard Deviation |  |
| Read Throughput Min |  |
| Read Throughput Max |  |
| Read Throughput Average |  |
| Read Throughput Standard Deviation |  |
| Availability |  |
| Concurrency |  |
| Successful Transactions |  |
| Failed Transactions |  |
| Longest Transaction |  |
| Shortest Transaction |  |
| Data Transferred |  |
| Cache Hit |  |
| Cache Miss |  |
| Cache Hit Ratio Int |  |

**CAP Theorem Analysis**

CAP Theorem Analysis C-Consistency A-Availability P-Partition Tolerance Ultimately to designate CAP Quality->CP, AP, etc…

NoSQL->Depth of JSON Nesting for Documents