UNIVERSITY OF CALIFORNIA
SANTA CRUZ

BENCHHUB: STORE DATABASE BENCHMARK RESULT IN DATABASE

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Pinglei Guo

June 2018

The Thesis of Pinglei Guo
is approved:

Professor Peter Alvaro, Chair

Professor Carlos Maltzahn

Professor Ethan L. Miller

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

BenchHub: Store database benchmark result in database

by

Pinglei Guo

Benchmark is an essential part of evaluating database performance. However, the procedure of setting up the environment and collecting result is time-consuming and not well defined. The uncertainty in benchmark procedure leads to low reproducibility. Furthermore, many benchmark results are highly compressed and only published in unstructured formats like document and graph. Without a structural format and the context of a benchmark, comparing benchmark results across sources is time-consuming and often biased.

In this thesis, BenchHub is presented to remedy those problems. It defines a job specification that covers the life cycle of running database benchmark in a distributed environment. A reference implementation of infrastructure is provided and will be hosted as a public service. Using this service, database developers can focus on analyzing benchmark result instead of getting them. BenchHub stores metrics like latency in time series databases and puts aggregated results along with benchmark context in relational databases. Users can query results directly using SQL and compare results across sources without extra preprocessing.

BenchHub integrates time series workloads like Xephon-B and standard database workloads like TPC-C. Comparisons between open source databases are made to demonstrate its usability. BenchHub is open sourced under MIT license and hosted on GitHub.

Acknowledgments

BenchHub is a combination of projects during my graduate study at UCSC. I worked with Zheyuan Chen on Xephon-B in CMPS232, Chujiao Hou helped me with Reika in CMPS203. My friends in UCSC helped me a lot in both coursework and daily life. I never missed homework until Ethan Vadai graduated before I do. Haiyu always comes to rescue when my laptop can't boot. Feimo and Yiming have been teleporting me across road 17 during the hardest time in my thesis.

Thanks to my family for supporting my graduate study and encouraging me to switch to a major I like. I was always making wrong decision on important things like choosing major in undergraduate study, but they never forced me to choose the way they thought is right.

The idea of the thesis and my interest in programming are mainly from my undergraduate study at Shanghai Jiao Tong University. Folks in Dongyue Web Studio allowed me to join regardless of my major and poor performance in interview. Without their guidance, I might still be writing AutoIt scripts and HTML with table layout. The experience in Prof. Cao's lab gave me the concept of time series database, which was the seed of my entire thesis.

I appreciate suggestions I got from my committee, especially Prof. Alvaro. The outline of BenchHub was formed in his distributed system class (CMPS232) as future work of the course project. He encouraged me to continue the work, and this thesis is the implementation.

Chapter 1

Introduction

Benchmark is vital for evaluating and comparing the performance of databases. However, many benchmark frameworks only provide the workload, and extra effort is needed to create the environment and keep track of results. The absence of environment provisioning automation makes running benchmark time consuming and hard to reproduce. This chapter gives a summary of existing database benchmarks and how we address their problems when designing BenchHub.

1.1 Motivation

There are two motivations for creating BenchHub. First, database benchmark is hard to run, and it is even harder for distributed databases. Second, benchmark results in existing publications cannot be used directly, they are unstructured and include little context for validating and reproducing.

1.1.1 Workload is just part of a benchmark

Historically, the primary difficulty of database benchmark was workload, the context of running workload got much less attention. A considerable effort was made on standardizing synthetic workload to reflect real-world application and not bias towards specific database vendors. TPC (Transaction Processing Performance Council) [13] was created in 1990 to stop database vendors from publishing unverified results as advertisements. YCSB [22] came out in 2010 to reflect the emerging use of distributed databases with non-relational model. However, those database benchmark workload standards only cover the workload itself (i.e., value distribution). There is no standard for how to set up databases and configure the operating system. The lack of constraint on benchmark context not only causes much extra work to users but also leads to some highly biased result.

To address the issue of running and reproducing distributed database benchmark, research efforts like YCSB++ [41], Oltpbench [24], TSDBBench [19] are made. However, as their names suggest, they are targeting a specific type of workload (NoSQL/RDBMS/TSDB) and can't be used for other types of databases. Furthermore, they prefer using ssh instead of node agent for simplicity and requires human interaction in different runs. This trait limits the extensibility and scalability of their system. Thus I did not build BenchHub on top of them.

To constrain the context of benchmark and make it reproducible, BenchHub defines a specification that covers the life cycle of a database benchmark and provides an infrastructure to run the spec. The spec contains selecting hardware, software installation, how to run the workload and collect results. It also defines an explicit syntax for declaring dependencies in

a distributed environment. Users only need the job spec if they want to validate or reproduce others' benchmark result.

1.1.2 Storing benchmark result in databases is better than plain text

The second problem of database benchmark is that the results are loosely compressed and presented in an unstructured format like document, graphs. This is caused by length requirement of publications. To address this issue, CERN started Zenodo [21] to share complete research data but there is no restriction on the format, users need to download huge files even if they want to query a small subset, extra documentation is needed for preprocessing downloaded data.

Unlike Zenodo, BenchHub chooses to store benchmark result in database with a schema. The schema is enforced by BenchHub when reporting result, users do not need to explicitly describe the format in the documentation. Databases have much higher compression ratio compared to plain text files and general compression like zip. Based on the characteristic of data and query patterns, time series data like query latency is stored in Time series database (TSDB) [28] for higher compression ratio; aggregated results are stored in relational database.

A drawback of storing results in databases is that the schema has to be defined in advance, which limits the number of workloads BenchHub supports. This tradeoff is reasonable because the number of standard database workloads is limited. Furthermore, BenchHub is an open source project hosted on GitHub; additional benchmark can be added by the community. Currently it supports three workloads, RDBMS (Oltpbench [24]), NoSQL (YCSB [22]) and TSDB (Xephon-B [30]).

1.2 Contributions and Outlines

There are three contributions. First, a job specification to describe database benchmark in a distributed environment. Second, a service to run benchmark based on spec and store results in databases. Third, integration with popular workloads for different types of databases.

The remainder of the thesis is organized as follows: Chapter 2 introduces BenchHub's job specification. Chapter 3 describes how we build the infrastructure to run the spec. Chapter 4 shows how to integrate BenchHub with standard database benchmark workloads. Chapter 5 demonstrates using BenchHub to compare the performance of various databases. Chapter 6 describes future work and Chapter 7 concludes the thesis.

Chapter 2

Job specification

Job specification defined by BenchHub covers the complete process of running database benchmark in a distributed environment. The spec is human-readable and used as instructions for running a benchmark on compatible infrastructure. In this chapter, we describe the spec and requirements on infrastructure. In next chapter, we will introduce the reference infrastructure implementation.

2.1 Challenges

2.1.1 Distributed execution

Continuous integration (CI) services like Jenkins [7] and Travis [14] use a single YAML file as job spec for running tests. However, they run on a single machine, and there is no way to explicitly specify distributed execution in their spec, users often write shell scripts to start container or virtual machine in the background to simulate a distributed environment

on a single node. This works for unit and e2e tests, but it is too inaccurate for benchmarking database. To avoid simulation, provision tools like Ansible [34] are used to ssh into machines during CI jobs to create and run benchmark in distributed environment. Those ad-hoc methods often introduce hard-coded configuration and require knowledge of specific provision tool to understand a benchmark, making the benchmark hard to reproduce.

BenchHub's solution is the node aware job specification, each stage in the job spec explicitly specifies nodes the tasks will run on. The spec is declarative, it does not have instructions for creating virtual/physical machine instances, this work is delegated to the underlying infrastructure. Another problem arises when the benchmark becomes distributed. If a job has multiple stages on different nodes, how to guarantee the dependencies between stages are met (i.e., databases should start before workload generators start loading data).

2.1.2 Dependencies in distributed system

There are many schedulers for running job in a distributed environment, i.e., Mesos [33], Kubernetes [20]. However, in their job spec, there is no explicit syntax for declaring dependency i.e., Service B get started if and only if service A is ready. This is also a problem even in local environment like docker-compose [36] when multiple services are used. Community members have created solutions like k8s-AppController [8] to solve dependencies between services in Kubernetes. Each service defines a `depend_on` field and the controller resolves it to a directed acyclic graph (DAG) before delegates it to the underlying executor.

BenchHub does not build the DAG using `depend_on` semantics, it asks users to construct the DAG explicitly in job spec using pipelines. Using `pipeline` instead of `depend_on`

reduces the burden of implementation and user mistakes like cycle dependencies. It also allows users to know the exact order of execution before submitting the job. Otherwise, they have to wait for the resolver to compute a generated DAG. Stages in a single pipeline are independent and execute in parallel, stages in next pipelines have to wait for the completion of all nodes involved in the current pipeline before it can start. For instance, if a workload generator B relies on database A, the stage for starting database A is put into the first pipeline, and workload generator is put into the second pipeline.

2.1.3 Readiness of a long running process

Another benchmark specific problem is making sure long running process like database server is ready. In routine tests, most processes are short running and exit code is enough to determine success or failure. However, when starting a database server, its parent (node agent, docker daemon) cannot tell if the child is running or still bootstrapping. Docker-compose solves it in local environment using scripts like wait-for-it, which blocks until database port is ready [1].

BenchHub enforces the readiness checks in spec by requiring all the task to specify if it is a long-running process. If it is a long-running process, then it must specify tasks to check its readiness. A long-running process is marked as complete when its readiness check is passed.

2.2 Elements

BenchHub's job specification contains four elements: node, pipeline, stage and task.

- node: select nodes and assign properties that are only visible to current job.

[1] `https://docs.docker.com/compose/startup-order/`

- pipelines: specify dependencies between stages and tasks
- stage: the basic unit for running multiple tasks on selected nodes
- task: the smallest unit of execution, a shell command or a container

2.2.1 Node assignment

In node section, users specify requirements like role, resources for nodes. Matched nodes are given names and labels, those aliases are only visible inside the job, users can refer to selected nodes in stage and task configuration using these aliases. There is only one node section in one job spec, it is used by the scheduler.

2.2.2 Pipeline

Pipeline exists in both stage and task, it is mandatory for stages but optional for tasks. Implicitly pipeline is generated for tasks in the order they are declared in the spec. Users should group stages that can be run in parallel (i.e., downloading binary) into one pipeline, and order pipelines based on dependencies (i.e., start database before run workload generator). There is one pipeline section for stages in one job spec, and one pipeline section for tasks in each stage.

2.2.3 Stage

Stage is the basic unit for running database and workload on multiple nodes, It contains two parts, node selectors and task groups.

2.2.3.1 Node selector

Node selector syntax is the same as node assignment, properties assigned in node assignment section can be used in node selector directly. The relationship between multiple selectors is OR, i.e., `[{name: db1}, {role: database}]` will select all the nodes that have `database` role or a name of `db1`.

2.2.3.2 Task group

Task group defines all the tasks that will be run on one node at one stage. Once all the tasks in task group are marked as complete, the stage on this node is marked as complete. A long-running process is marked as complete if the readiness check pass, it is an error if the process exits, even with 0 exit code.

All the selected nodes in one stage run same tasks. To proceed to next stage or finish the job, acknowledgments from all selected nodes in current stage is required, this is inspired by 2PC [27].

A stage must specify if it has long-running processes, although it is trivial to induct from task group, this extra verbosity is added to force users to be aware of long-running background processes.

2.2.4 Task

Task is the smallest execution unit, i.e., a shell command for workload generator, a docker container for database. It is recommended to use container for most tasks because resource constraint can be added while raw fork/exec gives the task same privilege as its parent

(the node agent). Long-running tasks require nested tasks to check readiness.

2.3 Example

A simple example is shown in Fig 2.1, it starts a sever in one node and pings the server in another node. First, two nodes are selected, server node and client node are given the name `srv` and `cli`. Then the first pipeline executes two stages `download_server` and `download_client` in parallel. After that, stage `start_server` is executed to start HTTP server on port 8080, the stage is marked as complete after `ready` task passed, which checks HTTP endpoint on server machine. Finally, client on another machine pings the server, server address is specified as template string in config. Config is rendered using current state of the job, server node is referred using its name `srv`.

2.4 Requirements for infrastructure

Based on job spec of BenchHub and characteristic of database benchmark, there are three requirements for infrastructure: a distributed environment, a scheduler that focuses on reducing resource race, and a node agent to run processes and collect metrics.

2.4.1 Real distributed environment

The spec allows user to run multiple workload generators on a cluster of databases. To get accurate result from benchmark, the test environment must be a real distributed environment. It cannot be simulated, even using a single powerful physical machine to spawn several virtual

```yaml
# start a server, wait for it, ping it
name: pingpong
reason: pingpong
framework: na
workload: na
database: na
nodes:
  - name: srv
    type: database
  - name: cli
    type: loader
stages:
  - name: download_client
    selectors:
      - name: cli
    taks:
      - driver: shell
        shell:
          command: "wget https://bh.io/pingclient-0.0.1.zip && unzip pingclient-0.0.1.zip"
  - name: download_server
      selectors:
        - name: srv
      taks:
        - driver: shell
          shell:
            command: "wget https://bh.io/pingserver-0.0.1.zip && unzip pingserver-0.0.1.zip"
  - name: start_server
    selectors:
      - name: srv
    tasks:
      - background: true
        driver: shell
        shell:
          command: "pingserver 8080"
        ready:
          - driver: shell
            shell:
              command: "waitforit -w http://localhost:8080/ping"
  - name: ping_server
    selectors:
      - name: cli
    tasks:
      - driver: shell
        shell:
          command: "pingclient http://{{.Nodes.srv.Ip}}:8080"
pipelines:
  - stages:
    - download_client
    - download_server
  - stages:
    - start_server
  - stages:
    - ping_server
```

Figure 2.1: BenchHub - Ping Pong job spec

machines [19] is still not ideal.

With the increasing popularity of public cloud service providers and decreasing price for on-demand resources, using a production-grade cluster of nodes is doable, especially for short running jobs like benchmarks. The infrastructure should be able to run on public cloud without special requirements on underlying hardware.

2.4.2 Scheduler without bin packing

A scheduler is needed because node assignment is not likely always a 1:1 mapping between resource at hand and job to be run. Many schedulers like Kubernetes [20] use bin packing, which put CPU intensive and IO intensive tasks on the same node to increase resource usage. However, this is not the case for database benchmarks. Both database and workload generators are resource-consuming, putting them on the same physical node (VM) causes too much noise and makes the result inaccurate. The goal of the scheduler used for benchmark should put reducing resource contention over increasing result utilization.

2.4.3 Node agent

Although it is possible to run a distributed benchmark using only ssh from a central machine without installing agents. It is not the right way to run BenchHub's spec. The spec allows executing tasks in parallel thus real-time feedback is needed, When using ssh, even with DevOps tools like Ansible [34], many ad-hoc scripts are needed to control node behavior and external service is needed for coordination. However, node agent can do it easily because it is deployed on the node and provides REST API or RPC calls for communication. Furthermore,

node agent can collect server metrics, which is an essential but often ignored part of benchmark results.

2.5 Related work

BenchHub's job spec is not the first one to describe complex execution in a distributed environment, similar specification already exists and is not limited to computer science. However, extra work is needed to fit them to database benchmark. Thus I created a more restricted spec to make implementation easier.

2.5.1 Popper

Popper [38] is created in UCSC to make system evaluation reproducible and provide a convention for use DevOps best practice for system researchers and the broader scientific community (bio, physics, HPC). It also provides a specification for performance evaluation. Compared to Popper, BenchHub's spec is limited to database benchmark instead of general systems, and it has more requirements on infrastructure. However, those limitations also allow implementation of BenchHub's infrastructure easier thanks to reduced features and tight coupling between components.

2.5.2 Workflow Language

Scientific communities have developed several workflow languages [17] [26] to make their research reproducible. However, a lot of components are designed for domain-specific problems, and database research is not covered. Also, the infrastructure and programming

models used by the scientific community are very different from internet companies. Scientific community uses HPC and MPI, which provide enormous resource and simple programming model. Internet companies use commodity hardware and scale horizontally. BenchHub mainly targets use cases of internet companies.

Chapter 3

Infrastructure

BenchHub defines a job spec for running database benchmark in a distributed environment. In this chapter, we present the reference infrastructure implementation. It is designed to be a multi-tenant system and can be deployed as public service. It contains a central node for scheduling and coordination and multiple worker nodes to spawn process and collect server metrics during the benchmark. Time series database is used to store dense metrics and aggregated results are put into relational databases.

3.1 Overview

BenchHub is designed to be a public service; users submit job using command line tools or Web UI. Based on the requirements in job spec, the scheduler selects idle nodes and starts running the job, nodes running database can no longer run other jobs, nodes running work loader generator may run multiple jobs. Database and workload generator will not be scheduled to same node. For single-tenant use, nodes are provisioned externally by the user. For public

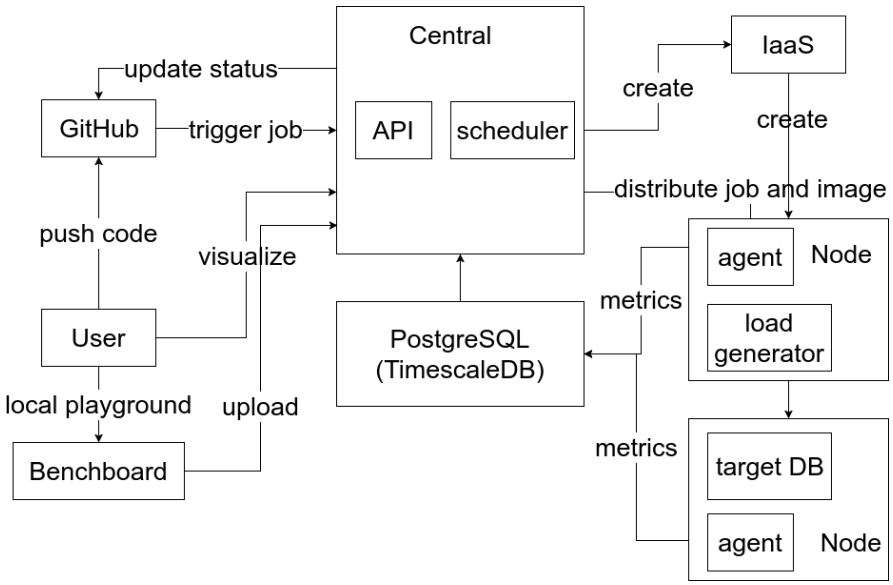

Figure 3.1: BenchHub - Public Service

service (Fig 3.1), BenchHub handles provisioning on public cloud service providers like AWS. With proper scheduling, a shared public service can reduce the overall cost by packing short running benchmarks into single paying period (i.e., 1 hour) and pick the cheapest provider.

The infrastructure of BenchHub has four components.

- Central, master node handling API request and scheduling.
- Agent, installed on worker nodes, runs tasks and collects metrics.
- Meta store, keeps cluster state, job specification and aggregated state in RDBMS.
- Time series store, stores metrics from both server and client during benchmark

The system is designed top down based on the requirements in job specification. Thus it is not intended to be used as or compared to general-purpose schedulers like Mesos [33].

3.2 Central

Central node is the control plane for managing node and scheduling job, it also provides API for command line tools and Web UI.

3.2.1 Node management

When an agent is started, it first registers itself to central and provides its node info including capacity, cloud provider (AWS, GCP, Packet). Central stores the node's info and gives it to the scheduler.

After register, agent would send heartbeat with its updated status (capacity, running jobs) at a fixed interval to central. If the central node has not received a heartbeat from node agent for a long time, it assumes the worker node is down and aborts all jobs running on that node.

3.2.2 Scheduler

Scheduler assigns nodes based on spec. After the nodes are assigned, control is given to job framework. Job framework runs the benchmark and returns the nodes to node scheduler after benchmark is finished. We are planning to support multiple framework in a two-level fashion like Mesos [33], but since there is only one framework, current implementation squashes the logic of node scheduler into job framework.

Node assignment is based on the three criteria defined in the `nodeAssignment` section of job spec: role, state and selector

- Role, nodes set their preferred role when start based on configuration during provision. Spec specifies the role the node would become if it is assigned. If there is an exact match, the node got higher ranking for this spec, loader node would never be scheduled to run database unless user starts two agents on the same machine with different config.

- State, nodes update their state when they are running jobs, only node in idle state can become a database node in the current job regardless of its role.

- Selector, nodes can have labels when they register themselves to central (i.e., `storage=ssd`) to specify capacity constraints beyond common properties like the number of CPU cores. Selector is also used later when scheduling job stages.

3.2.3 Planner

Planner generates execution plan based on node assignment and job spec. It is mainly used to help users understand the behavior of the system without actually executing it. It is inspired by Hashicoprp's Terraform[31], which we used to provision machines on different cloud service providers.

3.2.4 Provisioner

Provisioner uses IaaS API to scale up the cluster, it is designed for the public services, and is coupled with scheduler because short running benchmarks can be put into a single charge slot. Shrinking the cluster and finding the cheapest provider are also desired features.

Currently, we use Terraform [31] to spin up nodes because the scheduler is not multi

tenant. Vagrant is used for local development and testing because it runs on Windows with VirtualBox. However, Terraform is intended to be used as a command line tool instead of a library, its internal libraries do not guarantee compatibility. Future implementation of provisioner may use cloud service providers' SDK directly instead of using Terraform.

3.2.5 Executor

Executor uses the stage section in job config, it dispatches job to agents, and moves to next stage when the previous stage is finished. It requires successful execution of all selected nodes in one stage to mark this stage as finished. However, a finished stage does not mean all the tasks at that stage have stopped. Long-running tasks does not stop when its stage is marked as complete, they are mainly used for running databases in the background.

By having multiple stages, we solved the dependencies problem explicitly. It is a port of what people used for local environment in docker-compose [36], where one container must be ready before another container starts. Docker can start the process but can't tell if the process stalls or is already accepting connections, it requires scripts like `wait-for-it` to check ports. We followed the code in dockerize[1] and wrote a Go version of it, the binary is installed on all nodes.

However, using `wait-for-it` locally is not enough for a distributed environment, all loader nodes should be able to reach the database before benchmark starts. In BenchHub, it is recommended to have an extra stage to ensure loaders can reach the database. It is similar to 2 PC [27], where acknowledgment from all replicas are needed before moving forward. Users

[1] `https://github.com/jwilder/dockerize`

have to specify this stage explicitly, BenchHub does not insert it implicitly because it makes the spec harder to understand.

3.3 Agent

Agents are deployed on worker nodes, they are responsible for running processes and collecting server metrics. We plan to use them for distributing dataset and do map-reduce like computing in the future to increase resource utilization.

3.3.1 Runner

Runner runs commands based on the task section in job spec. Task also has pipelines like stages to run independent tasks in parallel. Runner supports three drivers: shell, exec and docker.

- Shell: command is passed to `sh -c` so variables in parameters will be expanded by shell, i.e., `echo $USER` would output *username*.
- Exec: command is run using fork and exec, parameters are passed as it is, i.e., `echo $USER` would output `$USER`.
- Docker: run a container or pull an image. Native docker-compose is not supported but can be simulated using multiple tasks.

3.3.2 Monitor

Monitor is implemented as a component in node agent and collects metrics during benchmark. Host metrics like CPU, Mem, Disk, Network usage are collected through `/proc` filesystem, container metrics are collected using docker API. Application specific metrics are collected using plugins.

Metrics collected during benchmark are tagged so they can be filtered out when looking at specific job or stage. We give the same data source multiple sets of tags (same machine, different jobs; same job, different stages etc.). This duplication is needed because major TSDBs do not allow the client to specify a unique id for series and tag it in multiple ways.

3.3.3 Distributor

Running and monitoring processes are not the only jobs for node agents, they can be used to distribute content in P2P. Dataset can be shared using torrent to save the bandwidth of central download server. Docker images can be distributed in P2P as well. Although docker's default registry is a single central server, companies like Alibaba has already implemented P2P image distribution in their Pounch engine[16] using Dragonfly[15] and is already deployed in production environment.

3.4 Storage

There are three storage components in BenchHub, meta store, benchmark result and time series.

3.4.1 Meta

Meta store keeps a global view of the cluster including node states, jobs status etc.. Current implementation uses central node's memory as a key-value storage and serializes objects using protobuf. Meta store has an interface, API server accesses meta store using the interface, and other components use API server. Thus switching to a more reliable meta store is transparent to other components.

3.4.2 Benchmark result

It is hard to have a general schema for benchmark result, current approach is creating a different schema for each benchmark frameworks. There aren't many benchmark frameworks, those supported by BenchHub are even fewer, so this approach does not have scale problem for now. Comparing across benchmark frameworks is not supported until we figure out a common subset of benchmark results.

3.4.3 Time Series

In YCSB [22] wiki it mentioned *while a histogram of latencies is often useful, sometimes a timeseries is more useful* The main drawback of how current benchmark tools output time series is they still write to text file or terminal in semi-structural format. Plain text result requires extra steps for visualizing and analyzing. Put time series data into time series database solved this, most of them come with visualization dashboard and query language. Also using a time series database significant reduce the space for storage, reducing the cost of maintaining BenchHub as a public service.

However, it is still hard to use TSDB for analyzing result directly beyond visualization. Most TSDBs query language is limited to select and aggregation, not to mention machine learning related functions. InfluxDB is working on Apache Arrow format[18] support in Go [2], which would allow working with python libraries directly without writing service to convert between different formats and "share" memory directly. It is possible to use Spark on time series databases directly in the future.

3.5 Implementation

BenchHub is written in Go [4]. The main reason for using Go is its balance between development speed and performance.

Current implementation of BenchHub (excluding UI and tests) is around 4000 lines of Go code. Communication between node uses GRPC, protobuf is also used for serialization meta data. For performance reason we use gogoprotobuf instead of official protobuf implementation. For communication with browser, HTTP JSON API is provided, it is not a 1:1 mapping with GRPC, only endpoints for visualization are provided. The Web UI is implemented using Angular and ant-design. UI pages dump JSON (Fig 3.2) instead of showing graphs and tables.

For running database, the recommended way is using Docker, this reduces the dependencies on worker node's linux distribution (windows is not supported). Volume and network need to be set properly to have reasonable result. BenchHub task runner is using official docker client API, most common parameters are supported. Docker daemon needs to be installed on the node before register unless node agent is running as root.

[2] `https://github.com/influxdata/arrow`

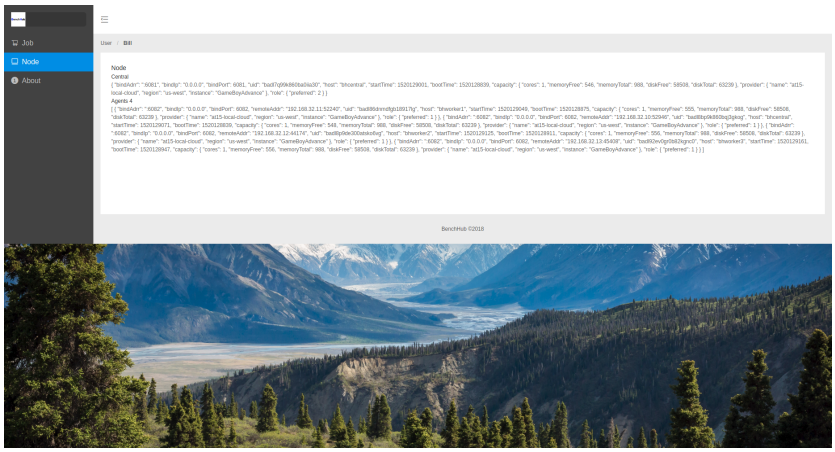

Figure 3.2: BenchHub: UI for node management

For data store, we use PostgreSQL and KairosDB [32] for relational data and time series data. Our time series database Xephon-K [29] is unstable, so we did not use it. Timescaledb was considered because it is a plugin for Postgres but it requires creating schema in advance. The metrics collected from server and clients varies based on workload and database, so we abandoned Timescaledb after a trial run. Because we use libtsdb-go [9] for writing to KairosDB, switching to a different time series database only requires configuration change on supported backends.

3.6 Limitation

The current implementation is not ready for public service, resilience and security are two major issues. The central node is a single point of failure, if it fails in the middle of a job, the behavior of worker nodes are undefined. In the future consensus services like Zookeeper [35], etcd [23] can be used to store the cluster state. We skipped them to speed up early stage

development, switching to those services only requires change in the API server and meta store.

Node agents are allowed to run arbitrary shell commands. If BenchHub is provided to the public, this is very dangerous, restriction like cgroup should be enabled by default. Also, users are allowed to run SQL query directly on results stored in relational databases, a translation layer (or proxy) is needed to allow only a subset of SQL on limited tables.

3.7 Related work and Discussion

BenchHub's infrastructure is greatly inspired by DevOps toolchains, continuous integration (Jenkins [7], Travis CI [14]), container orchestration (Kubernetes [20]), system monitoring (InfluxDB [6]). However our implementation is not a simple combination of popular solutions, core components like scheduler, node agents are written from scratch.

The main reason is for reinventing the wheel is we need fine-grained control so BenchHub can be extended to support fault injection. Most node agents are written to protect things instead of breaks things, latter is required in chaos engineering. Although chaos engineering is gaining popularity, it does not have first-class support in major platforms like Mesos, Kubernetes. By writing our own node agents, it is much easier to create chaos like partition nodes from network, use up all the inodes to test the resilience of the system.

3.7.1 Automated database benchmark

BenchHub is not the first one advocating fully automated database benchmarks. Many researchers have already taken this approach, but due to lack of maintenance, they are barely

usable now. YCSB++ [41] came out in 2011 to deploy and run YCSB in a distributed environment, but its pull request is still there after seven years. TSDBBench [19] uses Vagrant and python, it is actively maintained but requires extra work when using public cloud providers. OLTPBench [24] is maintained by CMU-DB group, but the distributed execution part is not usable last time we checked.

3.7.2 Load testing

Other than database benchmarks, there are many automated tools for general purpose load testing on web applications. Locust [10] contains a master slave setup for distributed load testing, it also allows configuring workload using python directly, but there is no way to specify the dependency in a distributed environment. Commercial tools like Soasta [12] and Flood.io [3] are powerful with colorful UI, however they are proprietary software and can't be extended. Mzbench [11] allocates nodes from EC2 directly, but it is for single user.

3.7.3 Continuous integration

CI tools like Travis [14] are typically used for testing correctness (unit test, e2e). Thus performance is not a major concern. For better resource usage, they run many jobs run on a single machine, lightweight isolation technology like Docker allows them to run even more jobs concurrently. However, this does not work for database benchmark, even workload generator could take a significant amount of resource. In the worst case a database may run alongside other databases. The parallelism in traditional CI services creates big noise when we could avoid it by limiting parallelism.

3.7.4 Jepsen

Although Jepsen [39] is used for correctness validation instead of performance evaluation, its framework is quite general, it automates database setup and runs databases under high load. A major difference of Jepsen is it injects faults and has expectations of what correct result should be. Jepsen contains control node and db nodes, *A Jepsen test runs as a Clojure program on a control node. That program uses SSH to log into a bunch of db nodes, where it sets up the distributed system*. Although this means fault injection can be done without agents on node, we want to collect database server machine metrics, so agent is still required for BenchHub.

3.8 Conclusion

We implemented a system for running database benchmarks and store results in databases based on BenchHub's spec. In next chapter, we show how to integrate this infrastructure with database workloads.

Chapter 4

Integration

BenchHub can be used for both developing databases and comparing databases performances. Popular workloads are integrated into BenchHub and can be used out of the box. This chapter describes required changes for both workloads and databases.

4.1 Workloads

Most workloads require a few changes to run on BenchHub because no human interaction is allowed during benchmark. For benchmark that writes results to plain text, a parser is needed to convert and write the data into database.

4.1.1 Oltpbench: Relational database

Oltpbench[24] is designed for transactional processing systems, mainly relational databases. It contains 15 workloads, TPC-C is well tested on BenchHub, other workloads may not work.

4.1.1.1 Change to workload

Integrating Oltpbench with BenchHub requires following changes:

- a catalog for supported databases
- utility for creating database
- parse text result and write to database

First, a catalog is needed because management interface is not consistent across RDBMS (i.e., `CREATE TABLE IF NOT EXISTS` will fail on PostgreSQL). We use a python script to read from the catalog and generate commands based on the current database, users do not need to care about the differences between databases.

Although Oltpbench can create tables, it can't create databases. There were people complaining on GitHub for database not found errors. The python script for reading catalog is also capable of calling database shell to create databases.

Results from Oltpbench are written in CSV files, we parse the files and write to database after benchmark is finished. Some results are not standard time series data (i.e. incremented id), so we store them in relational databases for now. We already submitted part of the changes in a pull request [1] to upstream.

4.1.1.2 Change to BenchHub

A table is created using the following schema to store the aggregated result from Oltpbench. Some attributes like scale-factor is not always presented because it is ignored by

[1] `https://github.com/oltpbenchmark/oltpbench/pull/241`

several workloads.

Name	Type	Description
database	varchar	name of database tested
version	varchar	version of database
workload	varchar	name of workload
config	text	raw xml config
scale-factor	int	number of points written
terminal	int	number of threads per node
worker_nodes	int	number of loader nodes
tps	int	number of transactions per second

Table 4.1: Oltpbench: Result schema in BenchHub

4.1.2 Xephon-B: Time series database

Xephon-B is a workload designed for time series database [37], time series database is also used by BenchHub to store metrics. An example configuration of Xephon-B is shown in figure 4.1.

4.1.2.1 Change to workload

Integrating Xephon-B with BenchHub requires the following changes. Xephon-B is maintained by the authors, so the changes are merged into master branch already.

- create database using database's API remotely
- report aggregated result to database

Although most time series databases are schema-less and usable out of box, some still requires creating database (i.e., InfluxDB), Creating database is added so Xephon-B can create database in a separated stage before loading to avoid human interaction.

Xephon-B supports writing metrics to time series database out of the box because it was designed to benchmark time series database. But aggregated results are simply echoed to stdout, the framework Xephon-B is using has driver for relational database, so it is modified to write to database directly, bypassing BenchHub API, this is subject to change in the future due to security and compatibility.

4.1.2.2 Change to BenchHub

On BenchHub side, Xephon-B is a shell task in job spec, configuration of workload and remote databases is passed using environment variables, environment variables are written as template in configuration and are rendered based on states in current job. Xephon-B itself can write result to database directly, so BenchHub creates a table in following schema and no parser is needed.

Name	Type	Description
database	varchar	name of database tested
version	varchar	version of database
workload	varchar	name of workload
workload_version	varchar	version of workload
config	text	raw xephon-b config
points	int	number of points written
series	int	number of series in single request
duration	int	time of loading phases
throughput	int	number of requests per second

Table 4.2: Xephon-B: Result schema in BenchHub

4.2 Databases

Most databases are already packaged into docker containers, the only extra effort for running database on BenchHub is defining the task for checking readiness of database server. BenchHub node agent ships with a helper binary called wait-for-it, it can listen to TCP port and exit with 0 if the port is ready or 1 if it has waited for too long.

4.2.1 Single node database

For single node databases, only one stage is needed to start database and run it in background. Figure 4.2 shows how to run a single node MySQL server on BenchHub. In `ready` task, `waitforit` connects to port 3306, which is the default port used by MySQL. Docker task driver maps port 3306 from container to host so loader generator can reach it using node's IP address, which is passed as parameters when rendering task templates.

4.2.2 Distributed database

For distributed databases, especially those require start up order, BenchHub's spec allow using multiple stages to model this behavior. Figure 4.3 is an example of setting a three node Cassandra cluster [40].

Node assignment acquires three nodes from idle nodes and give them name `first`, `second` abd `third`. In first stage, only one node is started, after this node is ready, we proceed to next stage. In second stage, two nodes are selected and ip address of first node is passed using template as seed for cluster, allowing two nodes to join the cluster. After both of the nodes are ready, the second stage is marked as complete.

```yaml
# example config of xephon-b
# select workload
workload: workload_0
# select database
database: influxdb_0
# select reporter
reporter: counter_0
# limit by
limit: time
#limit: points
duration: "10s"
worker:
  num: 10
workloads:
  - name: workload_0
    batch:
      series: 1 # only one series in one request
      points: 1 # only one 1 point in series
    series:
      prefix: "xbw0"
      num: 1 # number of series per worker
      churn: true # switch to new set of series after 5s
      churnDuration: "5s"
      numTags: 1
      groupPointsBySeries: false
    time:
      interval: "1ms"
      # TODO: noise
    value:
      generator: constant
#      generator: random
      constant:
        int: 1
        double: 12.3
      random:
        min: 0
        max: 100
        # TODO: distribution
databases:
  - name: influxdb_0
    type: influxdb
    influxdb:
      addr: http://localhost:8086
      database: xephonb
  - name: kairosdb_0
    type: kairosdb
    kairosdb:
      addr: http://localhost:8080
reporters:
  - name: counter_0
    type: counter
  - name: influxdb_0
    type: tsdb
    influxdb:
      addr: http://localhost:8086
      database: xephonbresult
```

Figure 4.1: Xephon-B config example

```
stages:
  - name: start_mysql
    selelectors:
      - role: database
    tasks:
      - driver: docker
        background: true
        ready:
          tasks:
            - driver: shell
              shell:
                command:
waitforit -w tcp://localhost:3306
        docker:
          image: mysql:5.7
          action: run
          env:
            - k: MYSQLPASSWORD
              v: "Dolphin"
          port:
            - guest: 3306
              host: 3306
```

Figure 4.2: Database: single node MySQL

```
nodeAssignments:
  - name: first
    role: database
  - name: second
    role: database
  - name: third
    role: database
pipelines:
  - name: start_first
    stages:
      - start_cassandra_first
  - name: start_rest
    stages:
      - start_cassandra_second
stages:
  - name: start_cassandra_first
    selelectors:
      - name: first
    tasks:
      - driver: docker
        background: true
        docker:
          image: cassandra:3.11
          action: run
          env:
            - k: CASSANDRA_BROADCAST_ADDRESS
              v: "{{.Node.Ip}}"
          port:
            - guest: 7000
              host: 7000
  - name: start_cassandra_second
    selectors:
      - name: second
      - name: third
    tasks:
      - driver: docker
        background: true
        docker:
          image: cassandra:3.11
          action: run
          env:
            - k: CASSANDRA_BROADCAST_ADDRESS
              v: "{{.Node.Ip}}"
            - k: CASSANDRA_SEEDS
              v: "{{.Nodes.first.Ip}}"
          port:
            - guest: 7000
              host: 7000
```

Figure 4.3: Database: Three nodes Cassandra Cluster

Chapter 5

Application

This chapter presents the result of running various workloads on BenchHub and potential usage of these results. Evaluation of RDBMS and Time Series Database (TSDB) are covered. Detailed hardware spec for results presented in this chapter can be found in Appendix A.

5.1 Track performance change of a database

Using BenchHub database developers can run and compare benchmark results of their system in different versions using one job spec. The results can tell if performance is degrading due to breaking changes in implementation.

We compared minor versions of MySQL (Fig 5.1) and PostgreSQL (Fig 5.2) using TPC-C workload in Oltpbench. The load generator is configured to use the default config with two client threads. Database servers run on 4 cores machine with 32 GB RAM and 120 GB SATA SSD, workload generators run on different machines (Appendix A). This is the default

setup for benchmarks in this chapter.

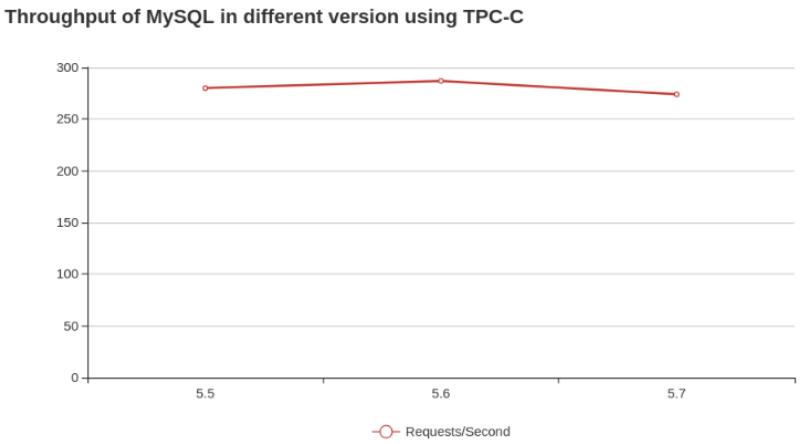

Figure 5.1: Oltpbench: Throughput of MySQL in different version

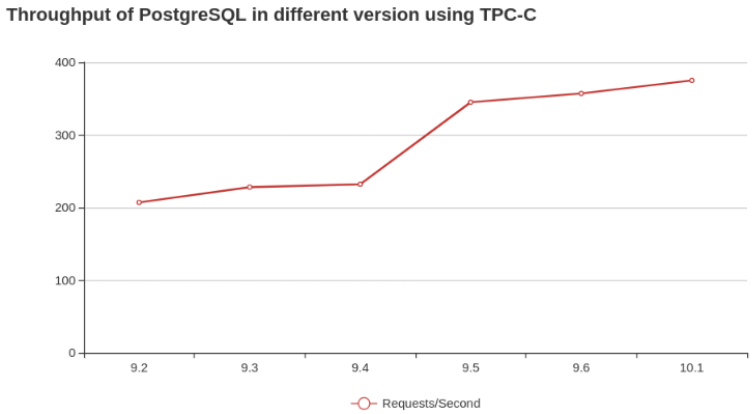

Figure 5.2: Oltpbench: Throughput of PostgreSQL in different version

The throughput of MySQL does not change much between minor releases, newest release (8.0) is not tested because Oltpbench's driver for MySQL is too old. PostgreSQL however, does show significant changes between different versions, version 10.1 almost doubled the throughput compared with version 9.2. Before 9.5, PostgreSQL's throughput is always lower

than MySQL's.

We also compared the throughput of InfluxDB in different versions using the time series workload (Xephon-B). In contrast to conventional assumptions, the throughput of InfluxDB does not increase with version number, it decreases in the middle (version 1.2) (Fig 5.3). However, in InfluxDB's developer blog, they said version 1.2 reduced lock contention and write performance increased 50% [1].

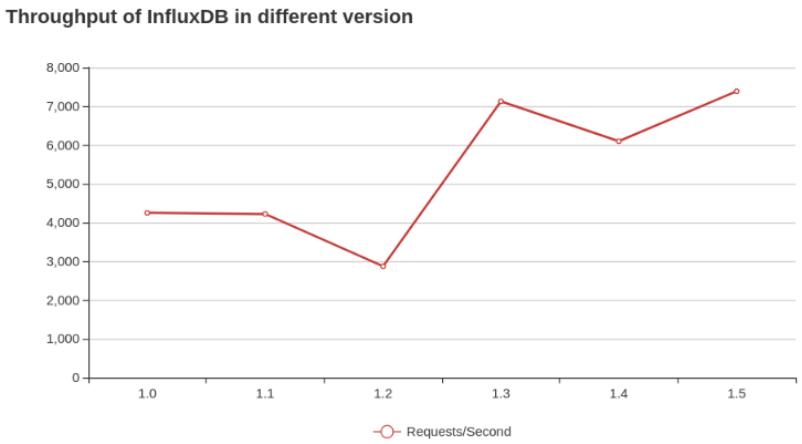

Figure 5.3: Xephon-B: Throughput of InfluxDB in different version

5.2 Compare performance of different databases

BenchHub can compare different databases directly because it stores results in databases. Results from different sources can be compared as long as the workload and hardware match.

We compared three time series databases, KairosDB [32], InfluxDB [6] and Graphite [5]. They are representatives for three popular storage backends, Cassandra, columnar store and

[1] InfluxDb 1.2 50 better write performance on larger hardware

round-robin file (RRD) respectively. The result is shown in Figure 5.4. To our surprise, Graphite is not slow regarding throughput, we were expecting Graphite to be the slowest because it is written in Python. However, it is faster than InfluxDB which is written in Go and KairosDB which is written in Java. One possible explanation is Graphite is using raw TCP while InfluxDB and KairosDB are using HTTP, when the payload is small, the overhead of HTTP protocol is significant. Another explanation is TCP clients do not wait for response, because some time series databases do not give response for write requests in TCP protocol. This finding results in more protocols being added to our time series database prototype Xephon-K [29]

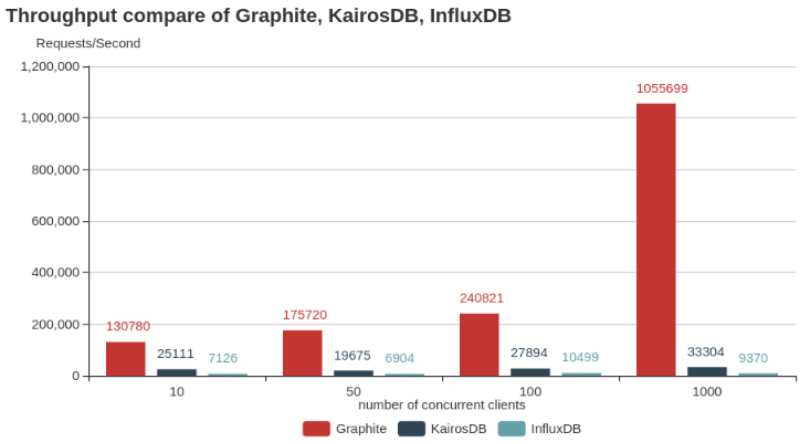

Figure 5.4: Xephon-B: Compare throughput of Graphite, KairosDB, InfluxDB

To validate the assumption that Graphite's throughput is from using TCP protocol, we run another benchmark. As shown in (Figure 5.5), all the databases use raw TCP. KairosDB supports both HTTP and Telnet, when KairosDB is using HTTP, Graphite's throughput is almost 10 times of KairosDB. However, after KaiorsDB switched to TCP, its throughput is 2 times of Graphite. Akumuli [1] is written in C++ and uses a redis like protocol, its throughput is much

higher than other databases.

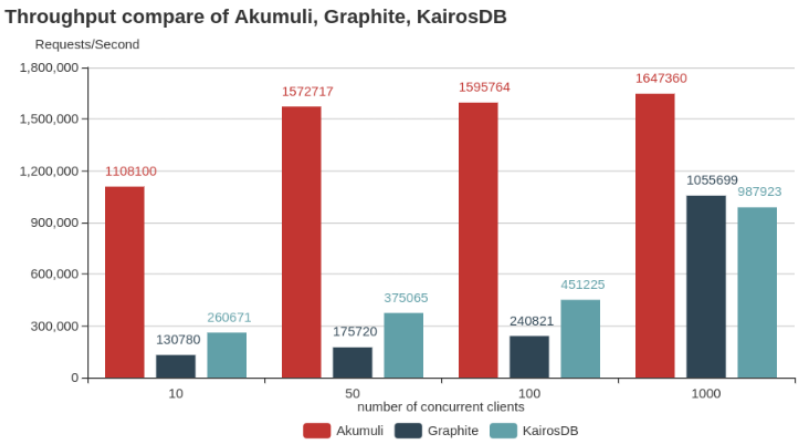

Figure 5.5: Xephon-B: Compare throughput of Akumuli, Graphite, KairosDB

5.3 Store latency result in time series database

Figure 5.6 shows client latency when benchmarking InfluxDB. Writing to time series database is more accurate and easier to use than summarizing via histogram, it does not require any pre-configuration or knowledge of value range. The reporter in workload generator writes metrics in full precision to time series database. Using a time series database also gives near real-time visualization of ongoing benchmarks.

5.4 Use server metrics for trouble shooting

Client side metrics can not tell everything about server while server side metrics can even tell about client. When running database using container, BenchHub can use cAdvisor [2]

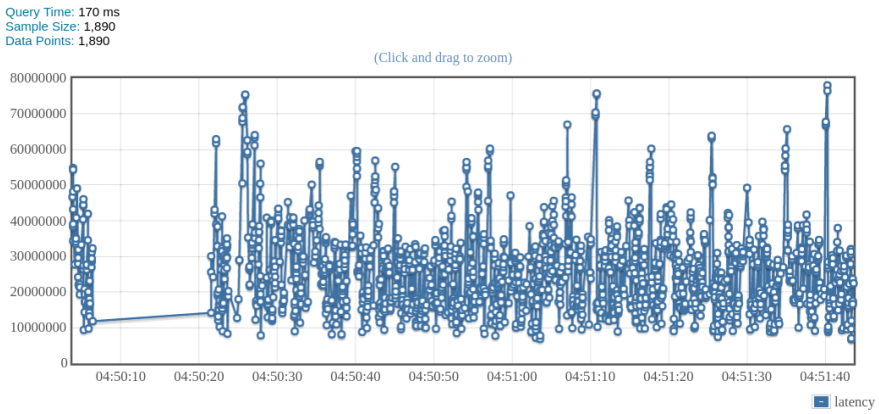

Figure 5.6: BenchHub: Write latency (ns) stored in time series database during benchmark

or docker API to collect metrics. cAdvisor comes with a default UI, so the graph can be viewed in realtime as shown in Figure 5.7 and 5.8.

One interesting observation is the periodical curves in server CPU usages. In our workload, we didn't limit QPS, the only thing that changes periodically is series identifier (series churn), it is used to simulate microservice deployment. However, the number of curves does not match the number of churns in benchmark, this reveals our benchmark (Xephon-B) is having bottleneck in itself that forces workers to stop generating load periodically. Later we found the reporter buffer is too small, once this buffer is filled up, workers are blocked until the buffer get drained. Without metrics collected on server sides, it is hard to detect this kind of anomaly using only client-side metrics.

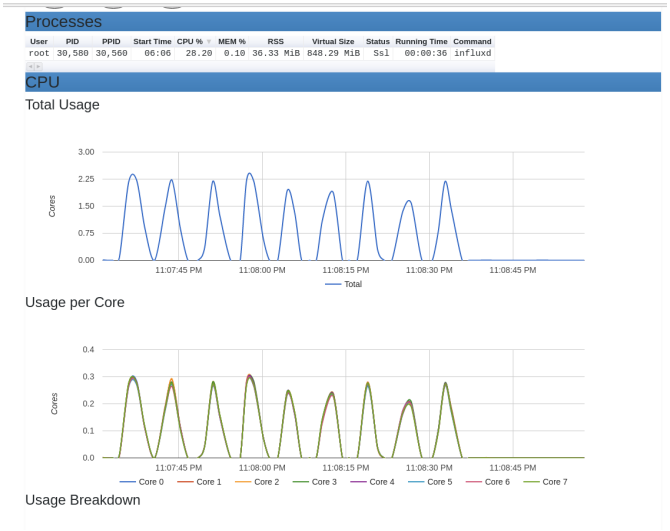

Figure 5.7: BenchHub: Real-time CPU usage during benchmark InfluxDB

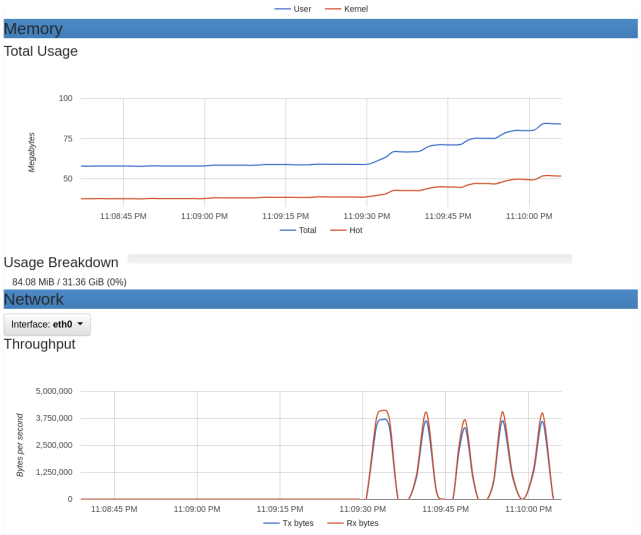

Figure 5.8: BenchHub: Real-time memory and network usage during benchmark InfluxDB

Chapter 6

Future work

6.1 Infrastructure

Current infrastructure implementation is a single tenant system, future development will mainly focus on the scheduler to make it multi-tenant.

Another direction is building BenchHub on top of existing systems like Kubernetes and Mesos, the job scheduling logic can be ported as a controller in Kubernetes or a framework in Mesos. This migration allows BenchHub to run along side other workloads without allocating dedicated hardware.

6.2 BenchBoard: a client for BenchHub

BenchHub requires setting up node agents and the central node, which is an overkill for microbenchmarks that can be run on a single machine. The spec for BenchHub can be adapted to support pure local execution, results are stored in user's home directory. SQLite

can replace PostgreSQL and Xephon-K's local storage [29] can replace dedicated time series database server. Furthermore, BenchBoard can synchronize data with BenchHub and serve as a local visualization tool.

6.3 Use machine learning for auto tunning

Using machine learning for tuning databases is becoming the new trend. Oracle announced their machine learning based tunning to reduce database maintenance cost, Google has published Vizier [25], CMU-DB group has published Ottertune [42]. One challenge for running machine learning using data from BenchHub is efficient data loading between BenchHub's data store and machine learning frameworks, Apache Arrow [18] is a possible solution for sharing memory. Another more aggressive approach is moving machine learning algorithm into database, which might scale better for larger data.

Chapter 7

Conclusion

BenchHub defines a job spec for running database benchmarks in distributed environment and provides the reference infrastructure implementation. Given the infrastructure, users only need a job spec to reproduce the benchmark. Benchmark results are stored in databases, allowing efficient query for both human and machine. BenchHub has builtin integrations with workloads like Oltpbench and Xephon-B. The integration process only requires a few changes in workloads and proper packaging on databases. Metrics collected by BenchHub can discover anomalies during benchmarks. With BenchHub, database developers can focus more on analyzing benchmark results instead of getting them. Database users can have more transparent performance comparisons thanks to the reduced cost of querying and reproducing benchmarks.

BenchHub is open sourced under MIT license, the code is available at `https://github.com/benchhub/benchhub`

Appendix A

Benchmark hardware

Most time series database I tested using BenchHub are not distributed, the default setup is just two nodes. One is a small machine (type 0) as loader and a regular machine (type 1) as database [1]. For more than one nodes setup, all database nodes are using type 1. All the nodes are in same VPC in one datacenter (SJ1).

Type 0 has 4 cores 2.4GHZ with 8GB DDR3 RAM, 80GB SSD and 1GPS network. Type 1 has 4 cores 3.5GHZ with 32GB DDR3 RAM, 120 GB SSD and 2GPS network. SSD brand is SAMSUNG_MZ7KM240.

`ulimit` for linux user running loader and database are both set to max to avoid issues like running out of file descriptors, other tweaks are set in cloud-init when provisioning machines and not written in job spec.

[1] `https://www.packet.net/bare-metal/`

Bibliography

[1] Akumuli. https://github.com/akumuli/Akumuli. Accessed: 2018-03-31.

[2] cadvisor. https://github.com/google/cadvisor. Accessed: 2018-03-31.

[3] Flood.io. https://flood.io/. Accessed: 2018-03-31.

[4] The go programming language. https://golang.org/. Accessed: 2018-03-31.

[5] Graphite. https://graphiteapp.org/. Accessed: 2018-03-31.

[6] Influxdb. https://github.com/influxdata/influxdb. Accessed: 2018-03-31.

[7] Jenkins. https://jenkins.io/. Accessed: 2018-03-31.

[8] Kubernetes appcontroller. https://github.com/Mirantis/k8s-AppController. Accessed: 2018-03-31.

[9] libtsdb. https://github.com/libtsdb. Accessed: 2018-03-31.

[10] Locust. https://github.com/locustio/locust. Accessed: 2018-03-31.

[11] Mzbench. https://github.com/satori-com/mzbench. Accessed: 2018-03-31.

[12] Soasta. https://www.soasta.com/load-testing/. Accessed: 2018-03-31.

[13] Transaction processing performance council. http://www.tpc.org/. Accessed: 2018-03-31.

[14] Travis ci. https://travis-ci.org. Accessed: 2018-03-31.

[15] Alibaba. Dragonfly. https://github.com/alibaba/dragonfly. Accessed: 2018-03-31.

[16] Alibaba. Pouch. https://github.com/alibaba/pouch. Accessed: 2018-03-31.

[17] Peter Amstutz, Michael R Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, et al. Common workflow language, v1. 0. 2016.

[18] Apache. Apache arrow. https://arrow.apache.org/. Accessed: 2018-03-31.

[19] Andreas Bader. *Comparison of time series databases*. PhD thesis, Diploma Thesis, Institute of Parallel and Distributed Systems, University of Stuttgart, 2016.

[20] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[21] CERN. Zenodo. `https://zenodo.org/`. Accessed: 2018-03-31.

[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[23] CoreOS. etcd: a dirstibued key-value store using raft. `https://github.com/coreos/etcd/`. Accessed: 2018-03-31.

[24] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[25] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017.

[26] GRAIL. Reflow. `https://github.com/grailbio/reflow`. Accessed: 2018-03-31.

[27] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[28] Pinglei Guo. Awesome time series databases. `https://github.com/xephonhq/awesome-time-series-database`. Accessed: 2018-03-31.

[29] Pinglei Guo. Xephon-k: A multi backend time series database prototype. `https://github.com/xephonhq/xephon-k`. Accessed: 2018-03-31.

[30] Pinglei Guo and Zheyuan Chen. Xephon-b: A time series database benchmark suite. `https://github.com/xephonhq/xephon-b`. Accessed: 2018-03-31.

[31] HashiCorp. Terraform. `https://www.terraform.io/`. Accessed: 2018-03-31.

[32] Brian Hawkins. Kairosdb. `https://github.com/kairosdb/kairosdb`. Accessed: 2018-03-31.

[33] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[34] Lorin Hochstein and Rene Moser. *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. " O'Reilly Media, Inc.", 2017.

[35] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[36] Docker Inc. Docker compose. `https://github.com/docker/compose`. Accessed: 2018-03-31.

[37] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2581–2600, 2017.

[38] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. The popper convention: Making reproducible systems evaluation practical. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 1561–1570. IEEE, 2017.

[39] Kyle Kingsbury. Jepsen: A framwork for distributed system verification, with fault injection. `https://jepsen.io/`. Accessed: 2018-03-31.

[40] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[41] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 9. ACM, 2011.

[42] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.