Reproducibility protocol for ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor search algorithms[⋆,⋆⋆]

Martin Aumüller[a], Erik Bernhardsson[b], Alexander Faithfull[a]

[a]*IT University of Copenhagen, Denmark*
[b]*Better, Inc., United States*

Abstract

In (Aumüller, Bernhardsson, Faithful, *Information Systems*, 2020), a benchmarking framework for nearest neighbor search implementations was introduced. The framework was used to evaluate a selection for nearest neighbor search algorithms on different datasets. This reproducibility companion paper details the experimental setup and provides a step-by-step description to reproduce the original results.

1. Introduction

Nearest neighbor search is one of the central techniques in many diverse areas of computer science such as image processing, recommender systems, data mining, and machine learning. The task of a nearest neighbor search algorithm is to preprocess a dataset $X \subseteq \mathbb{R}^d$ of n d-dimensional data points to answer nearest neighbor queries: Given a query point $x \in \mathbb{R}^d$, return the k nearest neighbors to x in X. While this can be efficiently solved for low-dimensional settings, such as $d \in \{2, 3\}$, exact algorithms often fall back to being similar (or worse) than a linear scan in high dimensions, a phenomenon called the "curse of dimensionality".

The present paper is a reproducibility companion paper of our primary paper [1], which introduces a benchmarking framework for implementations of nearest neighbor search algorithms called ANN-Benchmarks. [1] focused on a succinct description of the general approach of the framework, and presented an evaluation of state-of-the-art nearest neighbor search algorithms at the end of 2017 until the submission in mid-2018.

As discussed by Chirigati et al. in [2], many areas of science are in a reproducibility crisis. In particular in experimental computer science, there exist little

⋆DOI of original article: `https://doi.org/10.1016/j.is.2019.02.006`
⋆⋆Code repository: `https://github.com/maumueller/ann-benchmarks-reproducibility`
Research artifacts: `https://doi.org/10.5281/zenodo.4607761`

systematic effort to ensure soundness and reproducibility of experimental results. The contribution of the present paper is to provide an exact reproducibility protocol for benchmarking approximate nearest neighbor search algorithms, thus contributing to increasing reproducibility in this area. Firstly, it allows to reproduce our primary paper [1]. Secondly, it contains information on how to extend ANN-Benchmarks to include new algorithms and datasets, thus serving as starting point for future research on the topic.

Relation to current state of ANN-Benchmarks. This reproducibility companion paper details the exact steps needed to reproduce the plots in the paper [1]. Since the submission of the paper, many new algorithms were added to ANN-Benchmarks, and existing ones were refined. See `http://ann-benchmarks.com` for an up-to-date overview of nearest neighbor search implementations. While the current setup targets the reproduction of [1], the steps mentioned here are also valid for the more recent version of the benchmarking tool.

Experiments in Aumüller et al. [1]. We invite the reader to first take a look at [1] to get an idea about the scope of the framework. In a nutshell, we used our framework to compare many state-of-the-art nearest neighbor search algorithms on a broad collection of high-dimensional datasets. From each dataset, a certain collection of points was chosen as queries and presented to the algorithms.[1] Implementations were measured on their ability to quickly return a "good approximation" of the true nearest neighbors. Usually, this means that the throughput (measured in *queries per second*) was put into relation to the *average recall* (the average of the fraction of correct nearest neighbors among the query answers over all queries).

Results were reported on these performance/quality measures, but also on questions such as "how long does it take to build an index that will allow to achieve a recall of at least .9?" and "how adaptive are algorithms?"

A note to the reviewers. To speed up the reproducibility process, we would appreciate if problems are directly reported as issues via `https://github.com/maumueller/ann-benchmarks-reproducibility`.

2. ANN-Benchmarks framework overview

2.1. Code base

ANN-Benchmarks is primarily written in `Python`. Please refer to Table 1 for technical and legal information of the source code.

ANN-Benchmarks takes care of setting up, running, and evaluating a nearest neighbor search experiment. An experiment consists of running k-NN queries for a specified implementation on a predefined dataset. All implementations considered in [1] are listed in Table 2. One run consists of building the index for a list of

[1]For [1] those queries were chosen at random, but more refined approaches were later introduced in [3].

ANN-Benchmarks	Description
Github Repository	https://github.com/maumueller/ ann-benchmarks-reproducibility/
Release used in this work	`is_minor_revision1`
Legal code license	MIT
Source code languages	Python 3
Runtime requirements	Python at least 3.6, Docker version at least 1.41
Documentation	Source code and readme

Table 1: Technical and legal information of the latest version of the ANN-Benchmarks software library used in our experiments.

dataset points (using parameters related to *index building*), and running queries with parameters related to *query processing.* The authors of the individual implementations provided these parameter choices themselves, and ANN-Benchmarks just carries out the experiment using these parameters. The actual wrappers for the implementations are found in `ann_benchmarks/algorithms/`, a standard set of parameters can be found in `algos.yaml`.

2.2. An Overview over the Architecture

ANN-Benchmarks uses Docker to encapsulate different implementations. A conceptional overview over the architecture is given in Figure 1. This was a necessary step to allow easy handling of different implementations, which each one having its own dependencies. The Python runner invokes these Docker containers with the arguments necessary to run the experiment at hand. One container contains exactly one library tested by the benchmarking framework. Using Docker allows to limit the resources of each container, since we run all implementations single-threaded and enforce a limit on the memory usage. The main controller that manages all experiments lives outside the Docker environment and invokes different Docker containers based on the experiment that is run. During setup of the container, it mounts the Python module and the *data/* folder containing all datasets as read-only into the container to read the data, and mounts the *results/* directory as read-write to write the results back to the filesystem.

2.3. Dataset format

Table 3 gives an overview over the datasets considered in [1]. Generating these datasets is done by the script `create_datasets.py`, which internally calls `ann_benchmarks/dataset.py`. Each dataset is internally stored as an `hdf5` file, that contains the dataset points, the query points, the ids of the 100-nearest neighbors to each query point, and the distances of these points to the query. Since building this ground truth takes a considerable amount of time, all datasets are stored on `https://ann-benchmarks.com`. Before creating the dataset locally, the script always tries to download the `hdf5` file first. For the sake of reproducibility, all of the datasets are stored in the research artifacts detailed in Section 3.2.

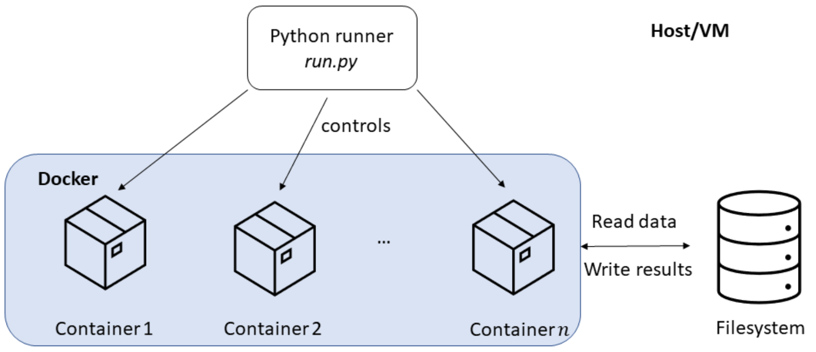

Figure 1: Conceptual overview over the architecture.

Principle	**Algorithms**
graph-based	`KGraph (KG)` [4], `SWGraph (SWG)` [5, 6], `HNSW` [7, 6]
	`PyNNDescent (NND)` [8], `PANNG` [9], `ONNG` [9, 10]
tree-based	*FLANN* [11], *BallTree (BT)* [6], `Annoy (A)` [12]
	`RPForest (RPF)` [13], `MRPT` [14]
LSH	*MPLSH* [15, 6]
other	`Multi-Index Hashing (MIH)` [16] (exact Hamming search),
	`FAISS-IVF (FAI)` [17] (inverted file)

Table 2: Overview of tested algorithms (abbr. in parentheses).

2.4. Result format

After finishing a run, a single `hdf5` file containing the results of the run is written to the file system in the *results/* folder. See Figure 2 for a partial snapshot of this directory. We stress that we write the *raw answer* of the query algorithm of the algorithm under consideration, that means the identifiers of the nearest neighbors returned for the individual queries. Only later on, this data is used to compute metrics such as (approximate) recall. The hierarchy it uses is *dataset/number_of_nearest_neighbors/algorithm/*. Each file contains the results of a single run (i.e., one set of query parameters). Each file contains general information of the run and details of the measurements, such as the identifiers of the nearest neighbors that were returned, query times, build time of the index, etc. The result file can be explored in the interactive Python console as shown in Figure 3.

2.5. Post-Processing results

After running the experiments, there are a couple of choices to visualize or process the data for another setting. The script `plot.py` creates a single PNG plot on a specific dataset for two given metrics, such as recall and throughput. `create_website.py` generates a website that visualizes *all runs* that can be found in the *results/* directory, split up by dataset (with varying algorithms)

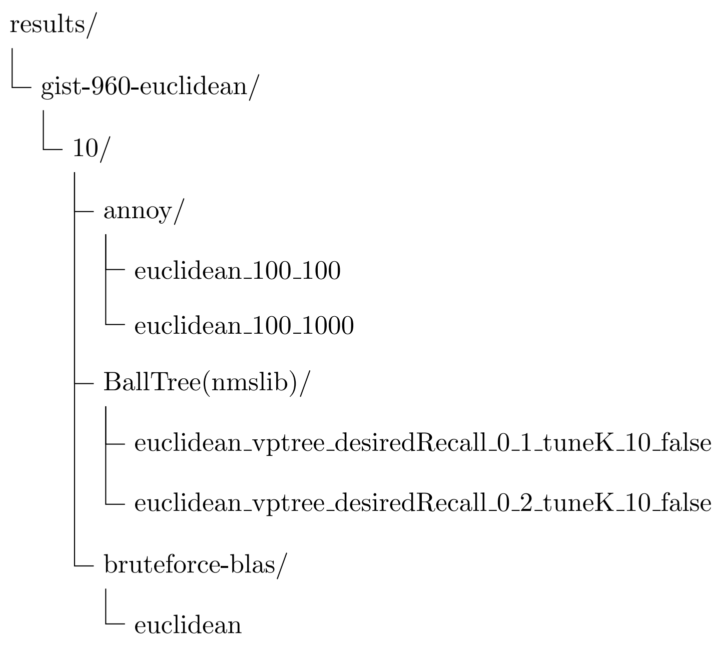

Figure 2: Overview over results in `results/` folder

```
>>> f = h5py.File("euclidean_reverse_1_true_100")
>>> dict(f)
{'metrics': <HDF5 group "/metrics" (3 members)>, '
   ↪ distances': <HDF5 dataset "distances": shape
   ↪  (1000, 10), type "<f4">, 'times': <HDF5
   ↪ dataset "times": shape (1000,), type "<f4">,
   ↪  'neighbors': <HDF5 dataset "neighbors":
   ↪ shape (1000, 10), type "<i4">}
>>> dict(f.attrs)
{'algo': 'kgraph', 'distance': 'euclidean', '
   ↪ run_count': 2, 'batch_mode': False, 'dataset
   ↪ ': 'gist-960-euclidean', 'build_time':
   ↪ 2678.368325471878, 'count': 10, 'name': '
   ↪ KGraph(euclidean)', 'best_search_time':
   ↪ 0.01496616005897522, 'index_size':
   ↪ 2022140.0, 'expect_extra': False, '
   ↪ candidates': 10.0}
>>> f["times"][:10] # individual query times of
   ↪ the first 10 queries
array([0.0064466 , 0.01282668, 0.00558615,
   ↪ 0.0106256 , 0.01036716, 0.005831  ,
   ↪ 0.00877213, 0.01525331, 0.00530338,
   ↪ 0.01109028],
      dtype=float32)
```

Figure 3: Structure of the HDF5 result file.

Dataset	**Internal Name**	**Data/Query**	**d**	**Metric**
SIFT	`sift-128-euclidean`	1 000 000/10 000	128	Eucl.
GIST	`gist-960-euclidean`	1 000 000/10 000	960	Eucl.
GLOVE	`glove-100-angular`	1 183 514/10 000	100	Cos.
NYTimes	`nytimes-256-angular`	234 791/10 000	256	Eucl.
Rand-Euclidean	`random-10nn-euclidean`	1 000 000/10 000	128	Cos.
SIFT-Hamming	`sift-256-hamming`	1 000 000/1 000	256	Ham.
Word2Bits	`word2bits-800-hamming`	399 000/1 000	800	Ham.

Table 3: Datasets under consideration.

or algorithm (with varying datasets). For reproducing [1], the main scripts are *data_export.py* and *reproducibility/generate_result_tables.py*, which exports the raw data to a csv file and generates data that can be plotted via *pgfplots*. The latter script also has special code to generate Figures 10 and 13 in [1].

2.6. Adding a new dataset

Adding a new dataset works by writing a Python function that takes care of downloading and parsing the original dataset into `numpy` arrays. ANN-Benchmarks takes care of computing the ground truth nearest neighbors automatically. The code for this has to be added to `ann_benchmarks/datasets.py` by adding the respective function that calls `write_output` as its final step, and adding the dataset in the bottom to the dictionary `DATASETS`. If runs on these datasets should be carried out, the dataset name has to be included in the for loops present in `reproducibility/run_experiments.sh`.

2.7. Adding a new algorithm

Adding a new implementation of an ANN algorithm requires to install the algorithm in a docker environment. If the library is called `XXX`, the installation goes into the file `install/Dockerfile.XXX` that inherits the *ann-benchmarks* base image. Next, a wrapper class that inherits from `BaseANN` has to be added to the wrappers in `ann_benchmarks/algorithms/`. Finally, an entry into the `algos.yaml` file has to be created that points to the constructor, the docker image, and contains the collection of parameters that should be tested. If runs with this implementation should be carried out, the yaml entries found in `reproducibility/` have to be edited to add the implementation with the parameter space that should be inspected. Most prominently, this has to be done in `reproducibility/standard_runs.yaml`.

3. The reproducibility experiments

This section describes the workflow to reproduce the experimental results from [1]. The original experiments were carried out on an Amazon EC2 c5.4xlarge instance, which was equipped with Intel Xeon Platinum 8124M CPU (16 available cores, 3.00 GHz, 25MB L3 Cache) and 32 GB of RAM using Amazon Linux.

Platform	Operating Sys.	Configuration	Tested by
Ubuntu 1$^{\dagger,\diamondsuit}$	Ubuntu 16.10	2x 14-core Intel Xeon E5-2690v4 2.60GHz, 512 GB RAM Quadro M4000, 6 TB HDD	Authors
Ubuntu 2†	Ubuntu 18.04	12-core Intel Xeon E5645 2.40GHz, 48 GB RAM, 2 TB HDD	Authors
			Reviewers
			Reviewers

Table 4: Detailed configurations of platforms used in the reproducibility study. † supports CPU-based experiments; ◇ supports GPU-based experiments.

Platform	Libraries & Env. Variables	Running time (CPU)	Tested by
Ubuntu 1	Python 3.6.5, Cuda 10.2 $PARALLELISM = 20$	4 days, 2 hours, 17 mins	Authors
Ubuntu 2	Python 3.6.5 $PARALLELISM = 3$ $GISTPARALLELISM = 1$	9 days, 27 mins	Authors
			Reviewers
			Reviewers

Table 5: Timings, libraries, and environmental variables used for reproducing the experiments.

Table 4 reports on the configurations used to reproduce the results of the experiment. Table 5 reports on the time it took to carry out the experiments, and the environmental variables used for running them on the specific machines. More details on the requirements to run all experiments can be found below.

Minimum requirements. To run all experiments, a modern (Intel) CPU with AVX-2 support, at least 20 GB of RAM, and a GPU supporting CUDA 10.2 or newer is required. If a GPU is not present, the GPU experiments cannot be carried out. If fewer than 20 GB of RAM are present, the dataset GIST-960-Euclidean cannot be processed. If fewer than 10 GB of RAM are available, the experiments cannot be run. If the CPU does not support AVX-2, the algorithm `ONNG` cannot run. At least 30 Gb of free space is necessary on a fresh installation to run all experiments.

3.1. General Overview

Figure 4 depicts the overall flow of the reproducibility protocol. There are two ways to carry out the reproducibility protocol. If an existing Linux-based machine is available, one can set up the framework following the guide in Section 3.3. Otherwise, a Vagrant box is available that sets up an Ubuntu 18.04 VM with a pre-installed framework. This method is covered in Section 3.4. After the installation succeeded, one can proceed to carry out the experiments as described in Section 3.5. After completing these experiments, raw results are processed into csv files and plots that allow reproducing the paper.

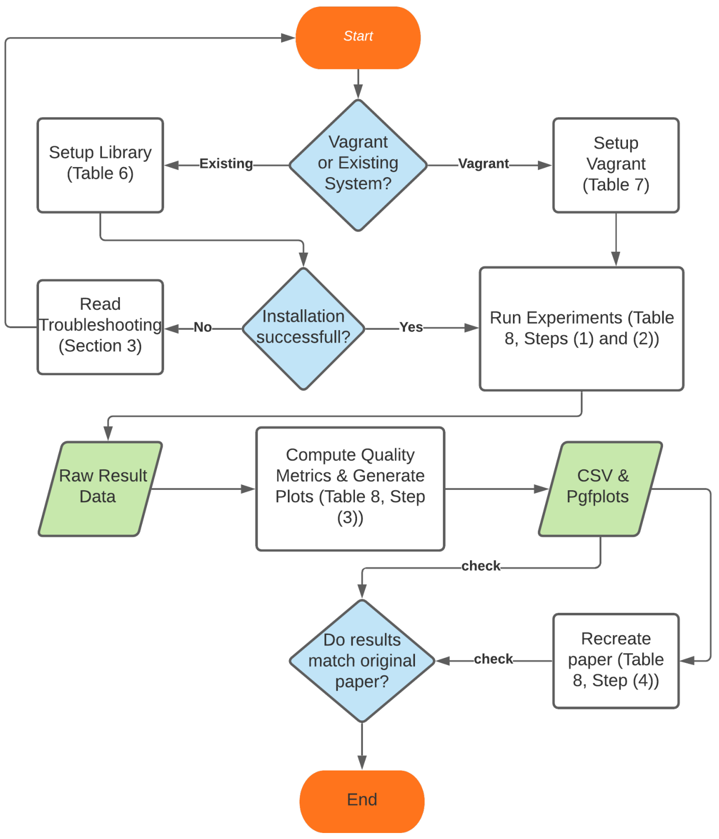
Start
Vagrant or Existing System?
Existing
Vagrant
Setup Library (Table 6)
Setup Vagrant (Table 7)
Installation successfull?
No
Yes
Read Troubleshooting (Section 3)
Run Experiments (Table 8, Steps (1) and (2))
Raw Result Data
Compute Quality Metrics & Generate Plots (Table 8, Step (3))
CSV & Pgfplots
check
Do results match original paper?
check
Recreate paper (Table 8, Step (4))
End

Figure 4: Overview over the reproducibility protocol. The framework can be installed on an existing system or using a Vagrant box that sets up a virtual machine with Ubuntu 18.04. Afterwards, experiments can be carried out. From these, quality metrics are computed and the original paper is reproduced. Detailed checks are possible via inspecting intermediate outputs, such as the csv file.

3.2. Research Artifacts

All research artifacts are provided in [18]. It fixes the version of the code used to produce the results in this reproducibility study. It also contains tar archives containing (i) all datasets used in the study, (ii) the original raw results used to produce [1], (iii) the raw results that we got from this reproducibility work, (iv) a *Vagrantfile* that spawns an Ubuntu VM ready to run all experiments, and (v) a tar file containing all binary Docker images. In these research artifacts, as well as in the Github repository at `https://github.com/maumueller/ann-benchmarks-reproducibility/`, the steps necessary to install, run, and evaluate the reproducibility protocol are documented, enabling easy copy-and-pasting in a more convenient way than from a PDF.

Step	Installation Guide for Ubuntu 18.04
(1)	Install docker
	$ sudo apt-get remove docker docker-engine docker.io containerd runc
	$ sudo apt-get update && sudo apt-get -y install apt-transport-https ca-certificates curl gnupg lsb-release
	$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
	$ echo ”deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable” | sudo tee /etc/apt/sources.list.d/docker.list >/dev/null
	$ sudo apt-get update && sudo apt-get install -y docker-ce docker-ce-cli containerd.io
	$ sudo usermod -aG docker $USER # logout and login again
(2)	Install nvidia-docker (GPU, requires working nvidia driver)
	$ distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
	&& curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -
	&& curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | sudo tee /etc/apt/sources.list.d/nvidia-docker.list
	$ sudo apt-get update
	$ sudo apt-get install -y nvidia-docker2
	$ sudo systemctl restart docker
(3)	Install Python 3.6, Git, LaTeX(for post-processing)
	$ sudo apt-get update
	$ sudo apt-get install -y python3-pip build-essential git texlive-fonts-extra texlive-science latexmk
(4)	Clone Github Repository, prepare docker images, setup datasets
	$ git clone https://github.com/maumueller/ann-benchmarks-reproducibility
	$ cd ann-benchmarks-reproducibility
	$ pip3 install -r requirements_py36.txt # requirements_py38.txt if running Python 3.8.
	$ python3 install.py --proc 5
	$ wget https://zenodo.org/record/4607761/files/data.tar?download=1 -O data.tar
	$ tar xf data.tar

Table 6: Installation guide on existing machine

3.3. Installation on an Existing Linux System

Installation. To install the software on an existing machine, a version of Linux with a Python version of at least 3.6 and a Docker version of at least 1.41 is required. For Ubuntu 18.04, the steps to set up a machine are detailed in Table 6. Step (1) in this table sets up Docker on the system. Step (2) sets up the GPU support for Docker containers in Ubuntu. Step (3) sets up a Python installation and LaTeX. These first three steps depend on the choice of distribution and will vary on existing systems. Step (4) sets up the local framework and will not vary. At the end of running the installation script `install.py`, the individual implementations will report on a successful or failed installation. It is necessary that all installations succeeded before proceeding.

Step	Installation Guide from Vagrantfile
(1)	Install Virtualbox and Vagrant on Ubuntu $ sudo apt-get update && sudo apt-get install -y wget virtualbox vagrant

(2)	Install Vagrant Box
	$ mkdir ann-benchmarks-reproducibility && cd ann-benchmarks-reproducibility
	$ wget https://zenodo.org/record/4607761/files/Vagrantfile?download=1 -O Vagrantfile
	Edit Line 51 and 52 in Vagrantfile to set suitable CPUs and RAM for the VM
	$ vagrant up
	$ vagrant ssh

Table 7: Vagrant guide

Troubleshooting.

- `pip3 install -r requirements_py36.txt` (or `pip3 install -r requirements_py38.txt`) does not succeed. If the local Python installation already has different versions of the necessary libraries installed, the installation might fail. In this case, `pip3 install -r requirements.txt` will try to install the dependencies without fixing library versions. If this does not work as well, we recommend creating virtual environments to start with a clean state for reproduction, as discussed `https://docs.python.org/3.8/tutorial/venv.html`.

- `python3 install.py --proc 5` reports failed installations. While we fixed all versions of the git repositories of the tested implementations, we do not control these repositories. If an installation fails, the research artifacts [18] contain binary images of the containers used for this reproducibility experiment. These can be loaded into docker via `docker load < docker-images.tar.gz`.

If these steps do not help, we recommend to set up a fresh VM using Vagrant as detailed in the next subsection.

3.4. Installation using Vagrant

We provide a *Vagrantfile* in the research artifacts discussed in Section 3.2 that automatically sets up a VM ready to carry out the experiments. A step-by-step installation guide for this case is given in Table 7. (The installation of Vagrant will be different on non-Ubuntu-based systems.) This setup does not allow to carry out the GPU experiments.

3.5. Running experiments

All the details to run the experiments and reproduce the paper are given in Table 8. Figure 5 provides a more detailed view on running and processing the experiments and the raw results.

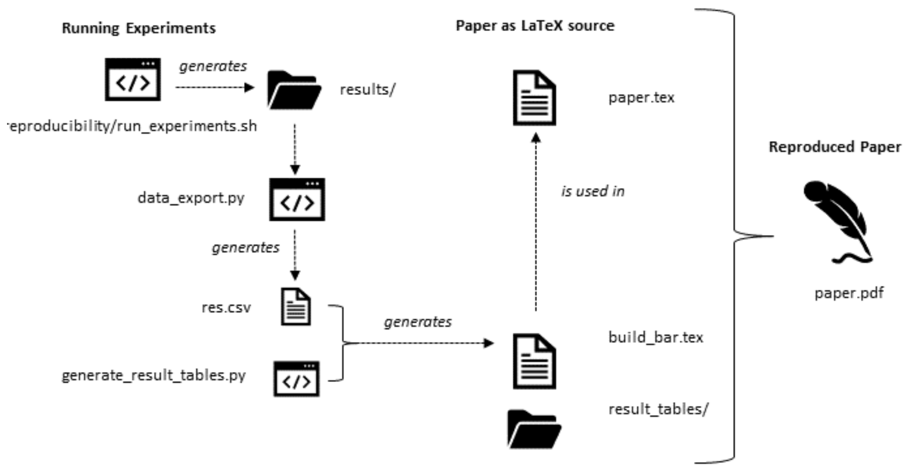

Figure 5: Overview of the reproducibility process in Table 8.

General comments. Before detailing the execution of the experiments, we provide the following general remarks.

- **Log files.** Running the experiments will give an high-level overview over the status of the experiments. Detailed logs are stored in `logs/`. Each dataset and k-NN combination, for $k \in \{10, 100\}$, creates exactly one log file.

- **Failures.** If experiments are interrupted, the framework will recover and only run those experiments for which it did not yet store results. Thus, the scripts can be re-run exactly as they are and do not have to be adapted.

- **Graceful degradation.** The framework gracefully handles unavailable installations and resource limitations, e.g., the amount of RAM available. It will fail on carrying out these experiments, as can be seen in the detailed logs, but will attempt to run all other experiments.

- **Detailed overview.** Appendix A contains detailed information about the amount of time certain experiments take and the amount of memory that is approximately necessary to carry out these experiments.

- **Adapting experiments.** The main bash script will take care of running

  ```
  $ python3 run.py --algorithm ALGO --dataset DATA --count
  [10, 100] [--batch]
  ```

  with the arguments to reproduce the paper. If certain runs should be left out, they can be removed from the bash script in `reproducibility/run_experiments.sh` or the yaml configuration files in `reproducibility/`, or the experiments can be started directly by invoking `python3 run.py`.

Step	Running Experiments and Reproducing Results
(1)	Running CPU-based experiment $ PY=python3 PARALLELISM=10 GISTPARALLELISM=3 bash reproducibility/run_experiments.sh | tee -a runs.log

(2)	Installing and running GPU-based experiment
	$ python3 install.py --algorithm faissgpu
	$ bash reproducibility/run_gpu.sh

(3)	Create output files
	$ sudo chmod -R 777 results && python3 data_export.py --out res.csv
	$ mkdir -p paper/result_tables/
	$ python3 reproducibility/create_result_tables.py res.csv paper/result_tables/
	$ python3 reproducibility/generate_and_verify_plots.py

(4a)	Produce LaTeX paper with working latex installation $ cd paper && latexmk -pdf paper.tex

(4b)	Produce LaTeX from Docker
	$ cd paper
	$ docker build . -t ann-benchmarks-reproducibility-latex
	$ docker run -it -v "$(pwd)"/:/app/:rw ann-benchmarks-reproducibility-latex:latest

Table 8: Running guide for experiments

Running CPU-based experiments. First, we run all CPU-based experiments by invoking:

```
$ PY=python3 PARALLELISM=10 GISTPARALLELISM=3 bash
    reproducibility/run_experiments.sh | tee -a runs.
    log
```

The environmental variable `PY` can be used to point to a custom Python 3.6 installation, e.g., provided by Anaconda. All individual runs of experiments in this part are carried out on a single CPU using Docker. The environmental variable `PARALLELISM` can be used to spawn multiple containers in parallel. On the machine Ubuntu 1, we used `PARALLELISM`=20. In general, around 10 GB of RAM are needed per process for most of the datasets. Thus, on a machine with 32GB of RAM, `PARALLELISM` can be set to at most 3. Note that for the largest dataset GIST-960-Euclidean, around 20GB of RAM are necessary per process, which meant in our setup that we had a peak memory usage of 400 GB. The environmental variable `GISTPARALLELISM` controls the number of parallel instances run for the GIST dataset. This was set to 20 as well on the Ubuntu 1 machine. On a machine with 32GB of RAM, `GISTPARALLELISM` must be set to 1. The script `run_experiments.sh` will report on the time it took to carry out all experiments.

Running GPU-based experiments. The paper [1] contains a single run of a GPU-based experiments in Figure 12. This run was carried out in a local

Step	Reproduce Results From Primary Paper

(1)	Getting raw results
	$ wget https://zenodo.org/record/4607761/files/results_original.tar?download=1 -O results.tar
	$ tar xf results.tar

(2)	Create output files
	$ sudo chmod 777 -R results/
	$ python3 data_export.py --out res.csv
	$ mkdir -p paper/result_tables/
	$ python3 reproducibility/create_result_tables.py res.csv paper/result_tables/
	$ python3 reproducibility/generate_and_verify_plots.py

(3a)	Produce LaTeX paper with working latex installation $ cd paper && latexmk -pdf paper.tex

(3b)	Produce LaTeX from Docker
	$ cd paper
	$ docker build . -t ann-benchmarks-reproducibility-latex
	$ docker run -it -v "$(pwd)"/:/app/:rw ann-benchmarks-reproducibility-latex:latest

Table 9: Reproduce paper from existing, raw results. Requires installation steps from Table 6 or Table 7 to be completed; the working directy is `ann-benchmarks-reproducibility`.

environment outside a docker container. To reproduce this run, we provide a script in *reproducibility/run_gpu.sh.* A Linux-based environment with a CUDA runtime of at least 10.2 is necessary. This can be checked by inspecting the output of `nvidia-smi`. Furthermore, the `nvidia-runtime` for Docker must be installed, as detailed in Table 6. If these requirements are met, the GPU run is reproduced by running:

```
$ python3 install.py --algorithm faissgpu
$ bash reproducibility/run_gpu.sh
```

Ubuntu 1 was equipped with a Quadro M4000 with compute engine 5.2 and all runs were finished within 10 minutes. If the reproducibility environment features an older GPU, the version of the compute engine must be manually set during compilation of FAISS in `install/Dockerfile.faissgpu` by editing the flag `DCMAKE_CUDA_ARCHITECTURES="75;72;52"`.[2]

3.6. Processing Raw Results

If all runs above have been carried out, we can start reproducing the plots in the paper. Run

```
$ sudo chmod 777 -R results/
$ python3 data_export.py --out res.csv
```

[2]An overview over the compute engines can be found on `https://developer.nvidia.com/cuda-gpus`.

```
$ mkdir -p paper/result_tables/
$ python3 reproducibility/create_result_tables.py res.
    csv paper/result_tables/
$ python3 reproducibility/generate_and_verify_plots.py
```

to create all the raw tables used by *pgfplots* during the final LaTeX compilation. Since exporting the results will compute all quality metrics, it took around 1 hour on our machine. (However, results are cached, so this cost applies only once.) All runs that have to be completed in order to build the paper are listed in Table 10. The script `generate_and_verify_plots.py` will generate the plot tex files necessary to compile the document. It will also list missing data points from Table 10, e.g., because the computation timed out, a too old CPU architecture was used, or no GPU was present. It will print the commands that can be used to directly re-run these experiments. However, the paper can be compiled even if files are missing, the respective lines in plots are then just omitted. Compile the paper by changing to the *paper* directory and compiling *paper.tex*, i.e.,

```
$ cd paper  && latexmk -pdf paper.tex
```

This requires a standard latex installation for scientific writing that was installed in Table 6. If such a system is not present, we provide another Docker container in *paper*. The reproducibility steps are then from within the *paper* directory. The final PDF can be seen in *paper/paper.pdf* and the plots can be compared to the original paper [1].

3.7. Comparison to Original Results

The result of the final step of the previous section is a version of the paper that is produced from the results obtained by running the experiments. A more detailed comparison can be achieved by comparing the csv files individually. We provide a Jupyter notebook `eval.ipynb` with some example comparisons in the Github repository.

3.8. Reproduction from the original raw results

To avoid rerunning all experiments, the raw result of the original runs can be accessed from the research artifacts (see Section 3.2). It is required to complete the *Installation* step in Table 6. Then, carry out the steps in Table 9.

3.9. Reflection on the Reproducibility Setup

Given the use of Docker in the ANN-Benchmarks setup, it proved difficult to provide a fully dockerized environment. We resorted to providing a VM image which uses docker internally. However, this makes it difficult to run the GPU-based experiments. On the other hand, ANN-Benchmarks comes with a very lightweight set of dependencies and is easy to install locally.

ANN-Benchmarks is a work in progress. Many parts of the benchmarking framework and the benchmarked implementations changed over time. This present reproducibility companion paper describes the steps to reproduce [1], but the very same setup works for producing all results on more recent versions.

Dataset	Count	Implementations
gist-960-euclidean	100	mrpt, annoy, SW-graph(nmslib), faiss-ivf, hnsw(nmslib), pynndescent
glove-100-angular	10	bruteforce-blas, BallTree(nmslib), hnsw(nmslib), pynndescent, annoy, SW-graph(nmslib), faiss-ivf, kgraph, flann, NGT-onng
	100	BallTree(nmslib), hnsw(nmslib), pynndescent, annoy, SW-graph(nmslib), faiss-ivf, kgraph, flann, NGT-onng
nytimes-256-angular	10	hnsw(nmslib), annoy, faiss-ivf
random-10nn-euclidean	10	pynndescent, annoy, SW-graph(nmslib), faiss-ivf, kgraph, hnsw(nmslib), NGT-onng
sift-128-euclidean	10	faiss-ivf-gpu-batch, BallTree(nmslib), hnsw(nmslib), pynndescent, annoy, SW-graph(nmslib), faiss-ivf-batch faiss-gpu-bf-batch, hnsw(nmslib)-batch, faiss-ivf, kgraph, flann, NGT-onng
	100	BallTree(nmslib), hnsw(nmslib), pynndescent, annoy, SW-graph(nmslib), faiss-ivf, kgraph, flann, NGT-onng
sift-256-hamming	10	annoy-euclidean, NGT-panng, annoy, faiss-ivf
word2bits-800-hamming	10	annoy-euclidean, NGT-panng, annoy, faiss-ivf

Table 10: Runs that need to finish before generating all plots.

In particular, we used the version of ANN-Benchmarks from January 2021 to reproduce the old results from 2017 and 2018. The main difficulty was in tracing the exact versions of the nearest neighbor search implementations in their GitHub repositories.

Of the time of writing, ANN-Benchmarks compares 26 different nearest neighbor search implementations while the experiments in this reproducibility companion paper used only 14. See `ann-benchmarks.com` for an up-to-date overview of nearest neighbor search algorithms.

4. Differences regarding our primary paper

Since most of our experiments consider the raw throughput achieved by the implementations, the compute architecture has a big influence on the individual plots. The throughput results on `Ubuntu 1` are roughly 1.5 to 2 times slower than the architecture used in the primary paper. However, general trends translate well into the new setting. Figure 6 and Figure 7 compare Figure 7 in [1] to the measurements on the machine used for reproduction. While absolute performance decreases, performance trends are comparable. We added two versions of [1] using the original results and the results from the reproducibility work to research artifacts discussed in Section 3.2.

We noticed the following differences between the reproduced plots and [1].

1. **Performance of `NND`.** The implementation of `PyNNDescent` [8] performed worse (in relation to others) on the new setup. We tried other (old) versions,

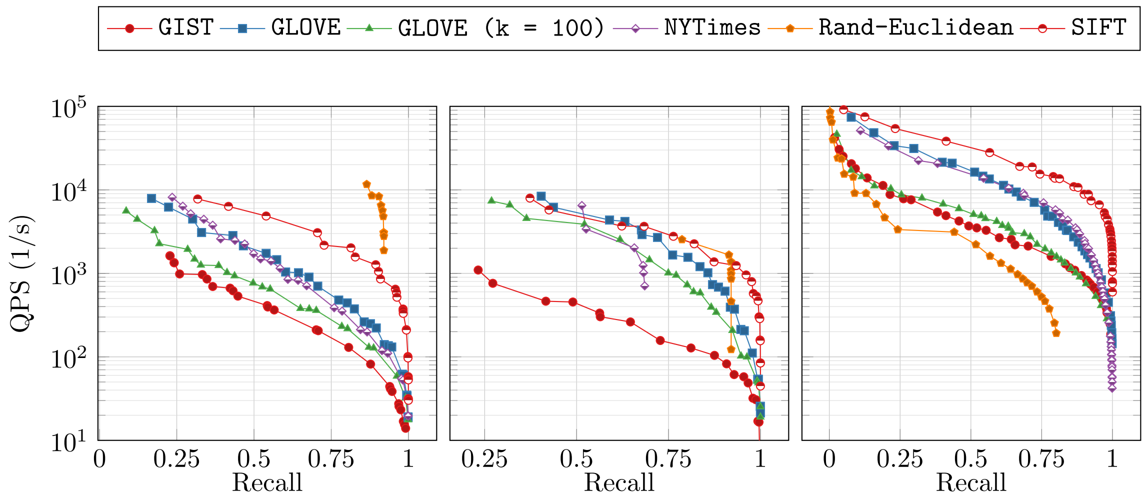

Figure 6: **Original:** Recall-QPS (1/s) tradeoff - up and to the right is better, 10-nearest neighbors unless otherwise stated, left: `Annoy`, middle: `FAISS-IVF`, right: `HNSW`. Recall measures the fraction of actual nearest neighbors among the returned 10 points of the implementation, averaged over 10 000 queries; QPS (queries per second) measures the time it took to answer these queries.

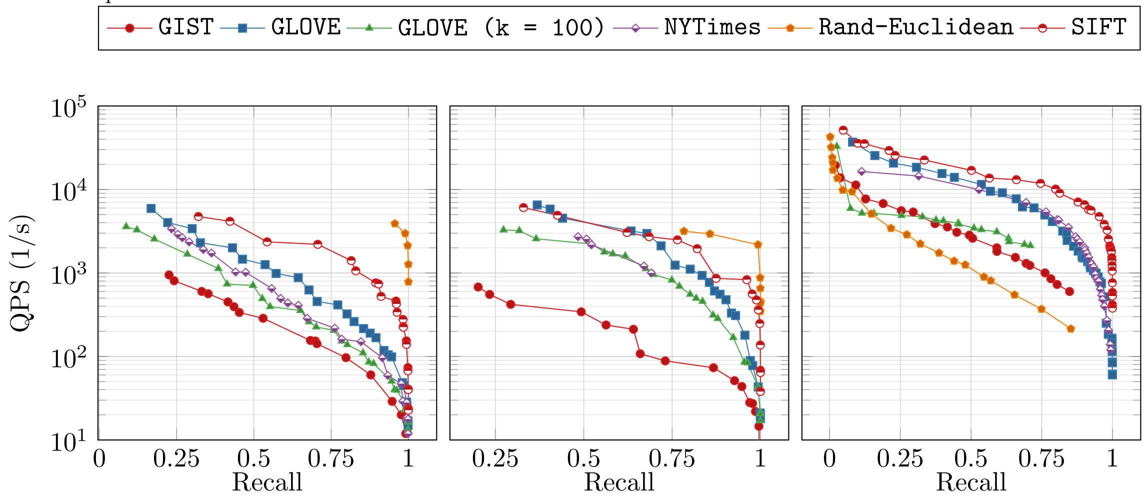

Figure 7: **Reproduced:** Recall-QPS (1/s) tradeoff - up and to the right is better, 10-nearest neighbors unless otherwise stated, left: `Annoy`, middle: `FAISS-IVF`, right: `HNSW`.

but the results were the same. More recent versions perform much better (being on par with HNSW in many cases), but we decided to report using an old version that is closer to the original performance.

2. **Omitted data points.** To improve the readability of the plots, we manually removed some data points in the original paper. For example, Figure 6 contained many data points with recall close to 1 which were removed. The reproduced version does not clean such data points.
3. **Differences in Figure 9.** `PANNG` is much faster in the reproducibility setup than in the original paper. This is because `PANNG` and `ONNG` are part of one library, and we had to use a more recent version to include `ONNG`. In the original paper, `PANNG` experiments where carried out in spring 2017, whereas the `ONNG` runs were done in autumn 2018. Furthermore, the line for `Annoy (eucl.)` on the plot to the left was wrong in the original paper. The performance is much better, as reported in the reproducibility experiment. One can see the mistake by a careful comparison between Figure 4 (bottom, left) and Figure 9 in [1].
4. **Longer build times.** We were not able to build indices that would allow for the same recall of HNSW on the reproducibility architecture. We increased the timeout to 12 hours (from 6 in the original paper) for an individual experiment.
5. **Differences in Figure 12.** The reproducibility machine has more cores and thus batch runs on the CPU are faster. On the other hand, its GPU is worse, so the GPU runs are slower. This means that the differences between CPU and GPU runs in Figure 12 are not as pronounced as in the original paper.
6. **Improvements in Performance in Reproducibility Setup.** Despite the performance differences inherent in the different CPU architectures, we also noticed that a few data point had improved on the slower architecture. This is because we set the timeout of individual experiments much higher, allowing for all indices to finish building. In the original paper, some of these runs timed out. However, this does not affect the conclusions drawn from the results. These differences can be seen in the Jupyter notebook `eval.ipynb` that is part of the Github repository.

References

[1] M. Aumüller, E. Bernhardsson, A. J. Faithfull, Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, Inf. Syst. 87 (2020).

[2] F. Chirigati, R. Capone, R. Rampin, J. Freire, D. Shasha, A collaborative approach to computational reproducibility, Information Systems 59 (C) (2016) 95–97.

[3] M. Aumüller, M. Ceccarello, The role of local intrinsic dimensionality in benchmarking nearest neighbor search, in: SISAP, Vol. 11807 of Lecture Notes in Computer Science, Springer, 2019, pp. 113–127.

[4] W. Dong, KGraph.
URL `https://github.com/aaalgo/kgraph`

[5] Y. Malkov, A. Ponomarenko, A. Logvinov, V. Krylov, Approximate nearest neighbor algorithm based on navigable small world graphs, Inf. Syst. 45 (2014) 61–68.

[6] L. Boytsov, B. Naidan, Engineering efficient and effective non-metric space library, in: SISAP'13, pp. 280–293.

[7] Y. A. Malkov, D. A. Yashunin, Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs, ArXiv e-prints (Mar. 2016). `arXiv:1603.09320`.

[8] L. McInnes, PyNNDescent.
URL `https://github.com/lmcinnes/pynndescent`

[9] Y. J. Corporation, NGT: PANNG.
URL `https://github.com/yahoojapan/NGT`

[10] M. Iwasaki, D. Miyazaki, Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data, ArXiv e-prints (Oct. 2018). `arXiv:1810.07355`.

[11] M. Muja, D. G. Lowe, Fast approximate nearest neighbors with automatic algorithm configuration, in: VISSAPP'09, INSTICC Press, pp. 331–340.

[12] E. Bernhardsson, Annoy.
URL `https://github.com/spotify/annoy`

[13] Lyst Engineering, Rpforest.
URL `https://github.com/lyst/rpforest`

[14] V. Hyvönen, T. Pitkänen, S. Tasoulis, E. Jääsaari, R. Tuomainen, L. Wang, J. Corander, T. Roos, Fast nearest neighbor search through sparse random projections and voting, in: Big Data (Big Data), 2016 IEEE International Conference on, IEEE, 2016, pp. 881–888.
URL `https://github.com/teemupitkanen/mrpt`

[15] W. Dong, Z. Wang, W. Josephson, M. Charikar, K. Li, Modeling LSH for performance tuning, in: CIKM'08, ACM, pp. 669–678.
URL `http://lshkit.sourceforge.net/`

[16] M. Norouzi, A. Punjani, D. J. Fleet, Fast search in hamming space with multi-index hashing, in: CVPR'12, IEEE, pp. 3108–3115.

[17] J. Johnson, M. Douze, H. Jégou, Billion-scale similarity search with gpus, CoRR abs/1702.08734 (2017).

dataset	count	batch	#experiments	#experiments left
glove-100-angular	10	false	57	533
sift-128-euclidean	10	false	63	470
random-10nn-euclidean	10	false	63	407
glove-100-angular	10	false	1	406
nytimes-256-angular	10	false	51	355
glove-100-angular	100	false	57	298
sift-128-euclidean	100	false	63	235
gist-960-euclidean	10	false	63	172
gist-960-euclidean	100	false	63	109
sift-256-hamming	10	false	35	74
word2bits-800-hamming	10	false	35	39
sift-128-euclidean	10	true	39	0

Table A.11: Experiments carried out by running `reproducibility/run_experiments.sh`.

[18] M. Aumüller, E. Bernhardsson, A. Faithfull, Research Artifacts for Reproducibility Paper "ANN- Benchmarks: A benchmarking tool for approximate nearest neighbor search".
URL `https://doi.org/10.5281/zenodo.4607761`

Appendix A. Detailed running times

Table A.11 collects the number of experiments carried out by running the CPU-based experiments. Table A.12 summarizes the running times for carrying out individual parts of the reproducibility protocol on a single thread. Each individual experiment can be re-run by invoking `python3 run.py` with the `--dataset` argument pointing to the dataset, and `--algorithm` pointing to the algorithm as labeled in the table. For example, running

```
$ python3 run.py --algorithm faiss-ivf --dataset gist
    -960-euclidean --count 10
```

will repeat the experiment in the row table below that tests `faiss-ivf` on the GIST dataset with 10-NN queries, and take roughly 3 hours to finish.

Note. The current size cannot be trusted because indices were built in parallel and the index size is estimated from the memory usage before and after building. (For example, notice some of the negative values.) We are currently re-running all experiments in a single thread. We will add these numbers to the next version.

algorithm	dataset	count	batch	Size (GB)	build (h)	total (h)
BallTree(nmslib)	gist-960-euclidean	10	False	3.77	7.93	7.95

Continued on next page

algorithm	dataset	count	batch	Size (GB)	build (h)	total (h)
BallTree(nmslib)	gist-960-euclidean	100	False	7.96	5.67	5.87
BallTree(nmslib)	glove-100-angular	10	False	1.36	2.32	2.35
BallTree(nmslib)	glove-100-angular	100	False	1.37	2.52	2.53
BallTree(nmslib)	nytimes-256-angular	10	False	0.82	5.81	5.83
BallTree(nmslib)	random-10nn-euclidean	10	False	1.98	4.72	4.77
BallTree(nmslib)	sift-128-euclidean	10	False	1.39	1.16	1.37
BallTree(nmslib)	sift-128-euclidean	100	False	1.39	1.11	1.13
NGT-onng	gist-960-euclidean	10	False	-1.97	5.06	5.23
NGT-onng	gist-960-euclidean	100	False	5.60	5.08	5.19
NGT-onng	glove-100-angular	10	False	2.31	1.46	1.50
NGT-onng	glove-100-angular	100	False	1.96	9.98	9.99
NGT-onng	nytimes-256-angular	10	False	1.16	6.21	6.23
NGT-onng	random-10nn-euclidean	10	False	2.96	6.08	6.10
NGT-onng	sift-128-euclidean	10	False	1.85	0.74	0.77
NGT-onng	sift-128-euclidean	100	False	0.80	1.29	1.38
NGT-panng	sift-256-hamming	10	False	1.99	0.38	0.46
NGT-panng	word2bits-800-hamming	10	False	1.73	1.86	1.87
SW-graph(nmslib)	gist-960-euclidean	10	False	3.14	5.00	5.06
SW-graph(nmslib)	gist-960-euclidean	100	False	4.11	3.86	4.01
SW-graph(nmslib)	glove-100-angular	10	False	1.54	1.53	1.54
SW-graph(nmslib)	glove-100-angular	100	False	1.54	1.57	1.58
SW-graph(nmslib)	nytimes-256-angular	10	False	0.56	0.38	0.39
SW-graph(nmslib)	random-10nn-euclidean	10	False	1.55	6.29	6.30
SW-graph(nmslib)	sift-128-euclidean	10	False	1.27	0.62	0.63
SW-graph(nmslib)	sift-128-euclidean	100	False	1.27	0.58	0.58
annoy	gist-960-euclidean	10	False	2.67	2.20	2.76
annoy	gist-960-euclidean	100	False	5.24	1.65	1.67
annoy	glove-100-angular	10	False	7.60	0.92	1.26
annoy	glove-100-angular	100	False	7.63	0.93	0.98
annoy	nytimes-256-angular	10	False	1.54	0.36	0.37
annoy	random-10nn-euclidean	10	False	6.83	1.63	1.79
annoy	sift-128-euclidean	10	False	5.77	0.68	0.69
annoy	sift-128-euclidean	100	False	5.77	0.58	0.58
annoy	sift-256-hamming	10	False	6.61	0.25	0.36
annoy	word2bits-800-hamming	10	False	2.15	0.10	0.22
annoy-euclidean	sift-256-hamming	10	False	6.70	0.69	0.70
annoy-euclidean	word2bits-800-hamming	10	False	3.31	0.43	0.43
bruteforce-blas	glove-100-angular	10	False	0.00	0.00	0.00
faiss-gpu-bf	sift-128-euclidean	10	True	0.63	0.00	0.01
faiss-ivf	gist-960-euclidean	10	False	4.80	3.14	3.14
faiss-ivf	gist-960-euclidean	100	False	5.00	1.73	1.89
faiss-ivf	glove-100-angular	10	False	0.65	0.26	0.38

Continued on next page

algorithm	dataset	count	batch	Size (GB)	build (h)	total (h)
faiss-ivf	glove-100-angular	100	False	0.66	0.26	0.26
faiss-ivf	nytimes-256-angular	10	False	0.41	0.23	0.26
faiss-ivf	random-10nn-euclidean	10	False	1.22	0.45	0.45
faiss-ivf	sift-128-euclidean	10	False	0.71	0.30	0.31
faiss-ivf	sift-128-euclidean	10	True	0.71	0.30	0.41
faiss-ivf	sift-128-euclidean	100	False	0.72	0.26	0.34
faiss-ivf	sift-256-hamming	10	False	1.40	0.43	0.45
faiss-ivf	word2bits-800-hamming	10	False	1.68	0.54	0.55
faiss-ivf-gpu	sift-128-euclidean	10	True	0.69	0.05	0.06
flann	gist-960-euclidean	10	False	4.18	9.01	9.07
flann	gist-960-euclidean	100	False	5.20	8.07	8.07
flann	glove-100-angular	10	False	0.78	7.73	7.86
flann	glove-100-angular	100	False	0.78	7.80	7.81
flann	nytimes-256-angular	10	False	0.40	0.14	0.14
flann	random-10nn-euclidean	10	False	2.03	11.86	11.86
flann	sift-128-euclidean	10	False	1.02	1.63	1.64
flann	sift-128-euclidean	100	False	1.02	1.68	1.68
hnsw(faiss)	gist-960-euclidean	10	False	1.89	25.91	26.14
hnsw(faiss)	gist-960-euclidean	100	False	4.15	26.24	26.24
hnsw(faiss)	glove-100-angular	10	False	1.39	17.04	17.04
hnsw(faiss)	glove-100-angular	100	False	1.39	16.43	16.44
hnsw(faiss)	nytimes-256-angular	10	False	0.47	14.28	14.37
hnsw(faiss)	random-10nn-euclidean	10	False	1.09	13.36	13.39
hnsw(faiss)	sift-128-euclidean	10	False	1.26	8.13	8.24
hnsw(faiss)	sift-128-euclidean	100	False	1.28	7.24	7.26
hnsw(nmslib)	gist-960-euclidean	10	False	4.35	5.77	5.85
hnsw(nmslib)	gist-960-euclidean	100	False	5.49	4.29	4.30
hnsw(nmslib)	glove-100-angular	10	False	4.64	22.41	22.41
hnsw(nmslib)	glove-100-angular	100	False	1.93	5.76	5.76
hnsw(nmslib)	nytimes-256-angular	10	False	0.83	6.84	6.84
hnsw(nmslib)	random-10nn-euclidean	10	False	3.26	8.95	9.18
hnsw(nmslib)	sift-128-euclidean	10	False	2.79	5.22	5.22
hnsw(nmslib)	sift-128-euclidean	10	True	2.79	0.53	0.53
hnsw(nmslib)	sift-128-euclidean	100	False	2.79	5.33	5.33
kgraph	gist-960-euclidean	10	False	1.57	1.24	1.40
kgraph	gist-960-euclidean	100	False	5.77	0.86	0.98
kgraph	glove-100-angular	10	False	13.22	3.35	3.35
kgraph	glove-100-angular	100	False	13.22	3.25	3.25
kgraph	nytimes-256-angular	10	False	3.80	1.49	1.50
kgraph	random-10nn-euclidean	10	False	2.13	0.59	0.60
kgraph	sift-128-euclidean	10	False	2.41	0.26	0.37
kgraph	sift-128-euclidean	100	False	2.41	0.24	0.24

Continued on next page

algorithm	dataset	count	batch	Size (GB)	build (h)	total (h)
mih	sift-256-hamming	10	False	1.47	0.42	0.50
mih	word2bits-800-hamming	10	False	6.54	0.34	0.44
mrpt	gist-960-euclidean	100	False	7.82	0.88	0.88
pynndescent	gist-960-euclidean	10	False	1.31	12.04	12.06
pynndescent	gist-960-euclidean	100	False	6.69	8.68	8.68
pynndescent	glove-100-angular	10	False	5.05	3.60	3.60
pynndescent	glove-100-angular	100	False	6.24	3.48	3.49
pynndescent	nytimes-256-angular	10	False	1.01	2.76	2.84
pynndescent	random-10nn-euclidean	10	False	6.31	14.80	14.80
pynndescent	sift-128-euclidean	10	False	4.69	4.71	5.54
pynndescent	sift-128-euclidean	100	False	5.13	4.29	4.95
pynndescent	sift-256-hamming	10	False	6.14	3.21	3.23
pynndescent	word2bits-800-hamming	10	False	3.98	1.88	1.89
rpforest	glove-100-angular	10	False	19.90	9.76	10.11
rpforest	glove-100-angular	100	False	19.31	9.64	9.66
rpforest	nytimes-256-angular	10	False	5.37	4.21	5.03
					Total:	**452h**

Table A.12: Summary of individual running time and memory requirements to carry out individual parts of the reproducibility framework.