ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms*

Martin Aumüller
IT University of Copenhagen, Denmark
maau@itu.dk

Erik Bernhardsson
Better, Inc.
mail@erikbern.com

Alexander Faithfull
IT University of Copenhagen, Denmark
alef@itu.dk

Abstract

This paper describes ANN-Benchmarks, a tool for evaluating the performance of in-memory approximate nearest neighbor algorithms. It provides a standard interface for measuring the performance and quality achieved by nearest neighbor algorithms on different standard data sets. It supports several different ways of integrating k-NN algorithms, and its configuration system automatically tests a range of parameter settings for each algorithm. Algorithms are compared with respect to many different (approximate) quality measures, and adding more is easy and fast; the included plotting front-ends can visualise these as images, LaTeX plots, and websites with interactive plots. ANN-Benchmarks aims to provide a constantly updated overview of the current state of the art of k-NN algorithms. In the short term, this overview allows users to choose the correct k-NN algorithm and parameters for their similarity search task; in the longer term, algorithm designers will be able to use this overview to test and refine automatic parameter tuning. The paper gives an overview of the system, evaluates the results of the benchmark, and points out directions for future work. Interestingly, very different approaches to k-NN search yield comparable quality-performance trade-offs. The system is available at `http://ann-benchmarks.com`.

2012 ACM Subject Classification H.3.3 Information Search and Retrieval

Keywords and phrases benchmarking, nearest neighbor search, evaluation

1 Introduction

Nearest neighbor search is one of the most fundamental tools in many areas of computer science, such as image recognition, machine learning, and computational linguistics. For example, one can use nearest neighbor search on labelled image datasets such as MNIST [?] to recognize handwritten digits, or one can find words that are semantically similar by applying the word2vec embedding [?] and finding nearest neighbors. The latter can, for example, be used to tag articles on a news website and recommend new articles to readers that have shown an interest in a certain topic. Other applications of nearest neighbor search include outlier detection [?], music classification [?], and recommender systems [?]. In some cases, a generic nearest neighbor search under a suitable distance or measure of similarity offers surprising quality improvements [?].

* The research of the first and third authors has received funding from the European Research Council under the European Union's 7th Framework Programme (FP7/2007-2013) / ERC grant agreement no. 614331. A conference version of this work was published at SISAP'17 and is available at `http://dx.doi.org/10.1007/978-3-319-68474-1_3`.

In many applications, the data points are described by high-dimensional vectors, with on the order of 100 to 1000 dimensions. A phenomenon called the *curse of dimensionality*, a consequence of several popular algorithmic hardness conjectures (see [?, ?]), tells us that, to obtain the true nearest neighbors, we have to use either linear time (in the size of the dataset) or time/space that is exponential in the dimensionality of the dataset. In the case of *massive* high-dimensional datasets, this rules out *efficient* and *exact* nearest neighbor search algorithms *in general*.

To obtain efficient algorithms, research has focused on allowing the returned neighbors to be an *approximation* of the true nearest neighbors. Usually, this means that the answer to finding the nearest neighbors to a query point is judged by how *close* (in some technical sense) the result set is to the set of true nearest neighbors.

There exist many different algorithmic techniques for finding approximate nearest neighbors. Classical algorithms such as kd-trees [?] or M-trees [?] can simulate this by terminating the search early, for example shown by Zezula et al. [?] for M-trees. Other techniques [?, ?] build a graph from the dataset, where each vertex is associated with a data point, and a vertex is adjacent to its true nearest neighbors in the data set. Others involve projecting data points into a lower-dimensional space using hashing. A lot of research has been conducted with respect to locality-sensitive hashing (LSH) [?], but there exist many other techniques that rely on hashing for finding nearest neighbors; see [?, ?] for a survey on the topic. We note that, in the realm of LSH-based techniques, algorithms guarantee sublinear query time, but solve a problem that is only distantly related to finding the k nearest neighbors of a query point. In practice, this could mean that the algorithm runs *more* slowly than a linear scan through the data, and counter-measures have to be taken to avoid this behavior [?, ?].

Given the difficulty of the problem of finding nearest neighbors in high-dimensional spaces and the wide range of different solutions at hand, it is natural to ask how these algorithms perform in empirical settings. In particular, while the general problem of finding nearest neighbors in high-dimensional datasets is difficult, high-dimensional datasets are often *not truly* high dimensional. That means that they are embedded into a high-dimensional space, but can be summarized in a space with much lower dimensionality. Measures such as the intrinsic dimensionality [?] or the more recent notion of local intrinsic dimensionality [?] try to capture this observation. Different techniques could have distinct ways to exploit the intrinsic lower dimensionality of many datasets.

Fortunately, many nearest neighbor search techniques have good implementations: see, e.g., [?, ?, ?] for tree-based, [?, ?] for graph-based, and [?] for LSH-based, solutions. This means that a new (variant of an existing) algorithm can show its worth by comparing itself to the many previous algorithms on a collection of standard benchmark datasets with respect to a collection of quality measures. What often happens, however, is that the evaluation of a new implementation is based on a small set of competing algorithms and a small number of selected datasets. This approach is bad for everyone: the authors of a new implementation do not get a representative view of how their work compares to others (and they, in turn, do not make it easy to use *their* work in such comparisons); reviewers and readers might not trust the reproducibility of these results, or the impartiality of the dataset, parameter and implementation choices; and potential users learn nothing useful about how an implementation actually compares with the rest of the field in practical conditions.

This paper proposes a way of standardizing benchmarking for nearest neighbor search algorithms, taking into account their properties and quality measures. Our benchmarking framework provides a unified approach to experimentation and comparison with existing work. The framework has already been used for experimental comparison in other papers [?] (to refer to parameter choice of algorithms) and algorithms have been contributed by the community, e.g., by the authors of NMSlib [?] and FALCONN [?]. An earlier version of our framework is already widely used as a benchmark referred to from other websites, see, e.g., [?, ?, ?, ?, ?].

Related work. Generating reproducible experimental results is one of the greatest challenges in many areas of computer science, in particular in the machine learning community. As an example, `openml.org` [?] and `codalab.org` provide researchers with excellent platforms to share reproducible research results.

The automatic benchmarking system developed in connection with the mlpack machine learning library [?, ?] shares many characteristics with our framework: it automates the process of running algorithms with preset parameters on certain datasets, and can visualize these results. However, the underlying approach is very different: it invokes whatever tools the implementations provide and parses their standard output to extract result metrics. Consequently, the system relies solely on the correctness of the algorithms' own implementations of quality measures, and adding a new quality measure would require a change in *every single* algorithm implementation. Very recently, Li et al. [?] presented a comparison of many approximate nearest neighbor algorithms, including many algorithms that are considered in our framework as well. Their approach is to take existing algorithm implementations and to heavily modify them to fit a common style of query processing, in the process changing compiler flags (and sometimes even core parts of the implementation). There is no general framework, and including new features again requires manual changes in each single algorithm.

Our benchmarking framework does not aim to replace these tools; instead, it complements them by taking a different approach. We only require that algorithms expose a simple programmatic interface for building data structures from training data and running queries. All the timing and quality measure computation is conducted within our framework, which lets us add new metrics without rerunning the algorithms, if the metric can be computed from the set of returned elements. Moreover, we benchmark each implementation as intended by the author without making fundamental changes of our own.

Contributions. We describe our system for benchmarking approximate nearest neighbor algorithms with the general approach described in Section 3. The system allows for easy experimentation with k-NN algorithms, and visualizes algorithm runs in an approachable way. Moreover, in Section 4 we use our benchmark suite to overview the performance and quality of current state-of-the-art k-NN algorithms. This allows us to identify areas that already have competitive algorithms, to compare different methodological approaches to nearest neighbor search, but also to point out challenging datasets and metrics, where good implementations are missing or do not take full advantage of properties of the underlying metric. Having this overview has immediate practical benefits, as users can select the right combination of algorithm and parameters for their application. In the longer term, we expect that more algorithms will become able to tune their own parameters according to the user's needs, and our benchmark suite will also serve as a testbed for this automatic tuning.

What do we benchmark? To some extent, it is fair to say that our system primarily benchmarks *implementations* rather than *algorithmic ideas* [?]; on the other hand, comparing sufficiently many different implementations might make it possible to draw reasonable inferences about the ideas they embody. Our evaluation, for example, includes three different implementations of a random projection forest and six implementations of k-NN graph-based algorithms; it shows clear differences between different implementations, but also a general difference between graph-based and random projection forest-based approaches.

2 Problem Definition and Quality Measures

We assume that we want to find nearest neighbors in a space X with a distance measure $\mathrm{dist}\colon X \times X \to \mathbb{R}$, for example the d-dimensional Euclidean space $\mathbb{R}^d$ under Euclidean distance (l_2 norm), or

Name of Measure	Computation of Measure
Index size of DS	Size of DS after preprocessing finished (in kB)
Index build time DS	Time it took to build DS (in seconds)
Number of distance computations	N
Time of a query	Time it took to run the query and generate result tuple π

Table 1 Performance measures used in the framework.

Hamming space $\{0,1\}^d$ under Hamming distance.

An algorithm $\mathcal{A}$ for nearest neighbor search builds a data structure $\text{DS}_\mathcal{A}$ for a data set $S \subset X$ of n points. In a preprocessing phase, it creates $\text{DS}_\mathcal{A}$ to support the following type of queries: For a query point $q \in X$ and an integer k, return a *result tuple* $\pi = (p_1, \ldots, p_{k'})$ of $k' \leq k$ distinct points from S that are "close" to the query q. Nearest neighbor search algorithms generate π by refining a set $C \subseteq S$ of candidate points with respect to q by choosing the k closest points among those using distance computations. The size of C (and thus the number of distance computations) is denoted by N. We let $\pi^* = (p_1^*, \ldots, p_k^*)$ denote the tuple containing the true k nearest neighbors for q in S (where ties are broken arbitrarily). We assume in the following that all tuples are sorted according to their distance to q.

2.1 Quality Measures

We use different notions of "recall" as a measure of the quality of the result returned by the algorithm. Intuitively, recall is the ratio of the number of points in the result tuple that are true nearest neighbors to the number k of true nearest neighbors. However, this intuitive definition is fragile when distances are not distinct or when we try to add a notion of approximation to it. To avoid these issues, we use the following distance-based definitions of recall and $(1+\varepsilon)$-approximative recall, that take the distance of the k-th true nearest neighbor as threshold distance.

$$\text{recall}(q, \pi, p_k^*) = \frac{|\{p \text{ contained in } \pi \mid \text{dist}(p,q) \leq \text{dist}(p_k^*, q)\}|}{k}$$

$$\text{recall}_\varepsilon(q, \pi, p_k^*) = \frac{|\{p \text{ contained in } \pi \mid \text{dist}(p,q) \leq (1+\varepsilon)\text{dist}(p_k^*, q)\}|}{k}, \quad \text{for } \varepsilon > 0.$$

(If all distances are distinct, $\text{recall}(q, \pi, p_k^*)$ matches the intuitive notion of recall.)

We note that (approximate) recall in high dimensions is sometimes criticised; see, for example, [**?**, Section 2.1]. We investigate the impact of approximation as part of the evaluation in Section 4, and plan to include other quality measures such as position-related measures [**?**] in future work.

2.2 Performance Measures

With regard to the performance, we use the performance measures defined in Table 1, which are divided into measures of the performance of the preprocessing step, i.e., generation of the data structure, and measures of the performance of the query algorithm. With respect to the query performance, different communities are interested in different cost values. Some rely on actual timings of query times, where others rely on the number of distance computations. The framework can take both of these measures into account. However, none of the currently included algorithms report the number of distance computations.

3 System Design

ANN-Benchmarks is implemented as a Python framework with several different interfaces: one script for running experiments and a handful of others for working with and plotting results. It automatically downloads datasets when they are needed and uses Docker build files to install algorithm implementations and their dependencies.

This section gives only a high-level overview of the system; see `http://ann-benchmarks.com` for more detailed technical information.

3.1 Algorithm implementations

Each implementation is installed via a Docker build file. These files specify how an implementation should be installed on a standard Ubuntu system by building and installing its dependencies and code. ANN-Benchmarks requires that this installation process also build Python wrappers for the implementation to give the framework access to it.

Adding support for a new algorithm implementation to ANN-Benchmarks is as easy as writing a Docker file to install it and its dependencies, making it available to Python by writing a wrapper (or by reusing an existing one), and adding the parameters to be tested to the configuration files. Most of the installation scripts fetch the latest version of their library from its Git repository, but there is no requirement to do this; indeed, installing several different versions of a library would make it possible to use the framework for regression testing.

At this point, we again emphasise that we are comparing algorithm *implementations*. Implementations make many different decisions that will affect their performance and two implementations of the same algorithm can have somewhat different performance characteristics [**?**]. When implementations expose other quality measures – such as the number of distance computations, which are more suited for comparing algorithms on a more abstract level – our framework will also collect this information.

Local mode. Using Docker is ideal for evaluating the performance of well-tuned implementations, but ANN-Benchmarks can also be used to help in the *development* process. To support this use case, the framework provides a *local mode*, which runs processes locally on the host system and not inside a Docker container. This makes it much easier to build a pipeline solution to, for example, automatically check how changes in the implementation influence its performance – in the standard Docker setup, each change would require the Docker container to be rebuilt.

Algorithm wrappers. To be usable by our system, each of the implementations to be tested must have some kind of Python interface. Many libraries already provide their own Python wrappers, either written by hand or automatically generated using a tool like SWIG; others are implemented partly or entirely in Python.

To bring implementations that do not provide a Python interface into the framework, we specify a simple text-based protocol that supports the few operations we care about: parameter configuration, sending training data, and running queries. The framework comes with a wrapper that communicates with external programs using this protocol. In this way, experiments can be run in external front-end processes implemented in any programming language.

The protocol has been designed to be easy to implement. Every message is a line of text that will be split into tokens according to the rules of the POSIX shell, good implementations of which are available for most programming languages. The protocol is flexible and extensible: front-ends are free to include extra information in replies, and they can also implement special configuration options that cause them to diverge from the protocol's basic behaviour. As the framework has

```yaml
float:
  euclidean:
    megasrch:
      docker-tag: ann-benchmarks-megasrch
      module: ann_benchmarks.algorithms.MEGASRCH
      constructor: MEGASRCH
      base-args: ["@metric"]
      run-groups:
        shallow-point-lake:
          args: ["lake", [100, 200]]
          query-args: [100, [100, 200, 400]]
        deep-point-ocean:
          args: ["sea", 1000]
          query-args: [[1000, 2000], [1000, 2000, 4000]]
```

Figure 1 An example of a fragment of an algorithm configuration file.

developed new operating modes, we have documented and implemented extensions to the protocol that allow it to take advantage of these modes.

The overhead imposed by using plain text representations is not entirely negligible; our benchmarks suggest that a linear search running in a subprocess using the protocol runs at about 75% of the speed of the same search called directly in the same process. On the other hand, the framework has operating modes that reduce this overhead virtually to zero, which we will discuss in Section 3.5.

Note, furthermore, that we have not been able to benchmark the overhead of other interfaces; it is quite likely, for example, that automatically generated wrappers, which construct many Python proxy objects to precisely mirror the underlying system, might impose a similar cost.

3.2 Datasets and ground truth

By default, the framework fetches datasets on demand from a remote server. These dataset files contain, in HDF5 format, the set of data points, the set of query points, the distance metric that should be used to compare them, a list of the true nearest $k = 100$ neighbours for each query point, and a list of the distances of each of these neighbours from the query point.

The framework also includes a script for generating dataset files from the original datasets. Although using the precomputed hosted versions is normally simpler, the script can be used to, for example, build a dataset file with a different value of k, or to convert a private dataset for the framework's use.

Most of the datasets use as their query set a pseudorandomly-selected set of ten thousand entries separated from the rest of the training data; others have separate query sets. The dataset file generation script makes this decision.

3.3 Creating algorithm instances

After loading the dataset, the framework moves on to creating the algorithm instances. It does so based on a YAML configuration file that specifies a hierarchy of dictionaries: the first level specifies the point type, the second the distance metric, and the third each algorithm implementation to be tested. Each implementation gives the name of its wrapper's Python constructor; a number of other entries are then expanded to give the arguments to that constructor. Figure 1 shows an example of this configuration file.

The `base-args` list consists of those arguments that should be prepended to every invocation of the constructor. Figure 1 also shows one of the special keywords, `"@metric"`, that is used to

pass one of the framework's configuration parameters to the constructor.

Algorithms must specify one or more "run groups", each of which will be expanded into one or more lists of constructor arguments. The `args` entry completes the argument list, but not directly: instead, the Cartesian product of all of its entries is used to generate *many* lists of arguments. Another entry, `query-args`, is expanded in the same way as `args`, but each argument list generated from it is used to reconfigure the query parameters of an algorithm instance after its internal data structures have been built. This allows built data structures to be reused, greatly reducing duplicated work.

As an example, the `megasrch` entry in Figure 1 expands into three different algorithm instances: `MEGASRCH("euclidean", "lake", 100)`, `MEGASRCH("euclidean", "lake", 200)`, and `MEGASRCH("euclidean", "sea", 1000)`. Each of these will be trained once and then used to run a number of experiments: the first two will run experiments with each of the query parameter groups `[100, 100]`, `[100, 200]`, and `[100, 400]` in turn, while the last will run its experiments with the query parameter groups `[1000, 1000]`, `[1000, 2000]`, `[1000, 4000]`, `[2000, 1000]`, `[2000, 2000]`, and `[2000, 4000]`.

3.4 The experiment loop

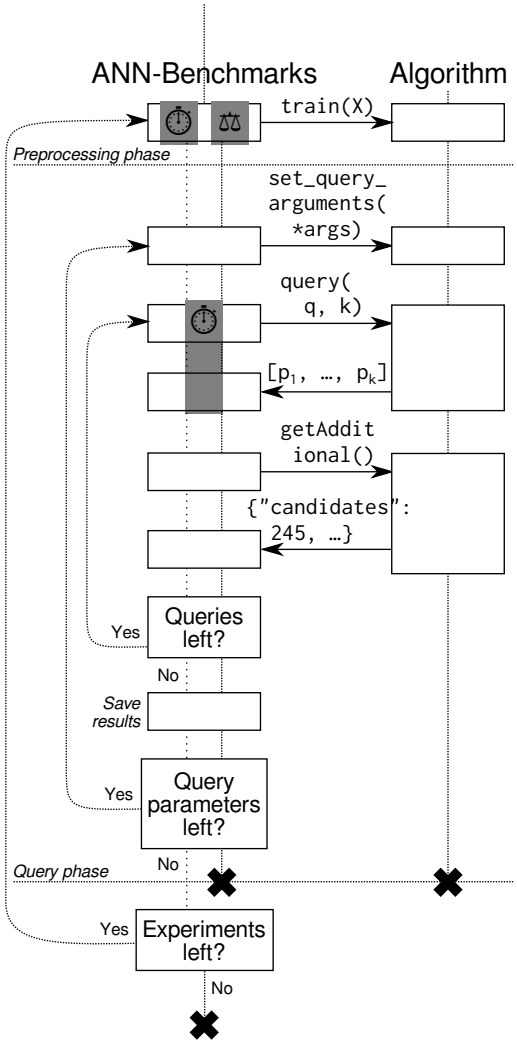

Figure 2 Overview of the interaction between ANN-Benchmarks and an algorithm instance under test.

Once the framework knows what instances should be run, it moves on to the experiment loop, shown in Figure 2. The loop consists of two phases. In the *preprocessing phase*, an algorithm instance builds an index data structure for the dataset X, while the framework records how long this takes and how much additional memory it uses. The loop then transitions to the *query phase*, in which query points are sent one by one to the algorithm instance, while the framework records how long the responses take to arrive. For each query point, the instance returns (at most) k data points; after answering a query, it can also report any extra information it might have, such as the number of candidates considered, i.e., the number of exact distances computed. The instance is then reconfigured with a new set of query parameters, and the query set is run repeatedly, until no more sets of these parameters remain.

Each algorithm instance is run in an isolated Docker container. This makes it easy to clean up after each run: simply terminating the container takes care of everything. Moving experiments out of the main process also gives us an implementation-agnostic way of computing the approximate memory usage of an implementation: the subprocess records its total memory consumption before and after initialising the algorithm instance's data structures and compares the two values. (This value is necessarily only an rough estimate – in particular, it cannot

take into account garbage collection, or processes that use and then attempt to release lots of temporary memory during the training process – but it is nevertheless useful to have.)

The complete results of each run are written to the host by mounting part of the file system into the Docker container. The main process performs a blocking, timed wait on the container, and will terminate it if the user-configurable timeout is exceeded before any results are available.

Dataset size. In its current form, ANN-Benchmarks only supports benchmarking *in-memory* nearest-neighbor algorithms. In particular, the dataset is kept in memory by ANN-Benchmarks when running experiments. This has to be taken into account when choosing datasets to include into the framework. In practice, we need a machine with 32GB of RAM to run all the experiments, where the largest dataset has around 1 000 000 items consisting of 1 000 dimensions each.

3.5 Batched queries and multi-threading

Running queries one by one is not necessarily representative of real-world query workloads; many systems allow queries to be batched together. ANN-Benchmarks has an alternative operating mode, *batch mode*, in which all the queries are passed, and all the results are returned, at once. This enables many interesting approaches to parallelization that would not be possible when running single queries.

When not running in batch mode, ANN-Benchmarks locks the Docker container that hosts the experiment loop to a single thread on a single CPU, using the Linux kernel's `cpusets` capabilities to restrict access to the system's resources. This is intended to make comparisons fairer: without this lock, the structure of the loop would give an implementation that ran a single query across multiple threads an unfair advantage over an implementation designed to use multi-threading to run several queries at once. In batch mode, all of the host system's resources are made available to the Docker container.

The behaviour of the experiment loop diverges slightly from Figure 2 in batch mode. Batch queries do not return a sequence of tuples containing answers to the individual queries; instead, these results are obtained via an additional method, akin to Figure 2's `getAdditional()` method. This allows an algorithm to return the result of a batch query as an opaque internal data structure; this will stop the clock, and the additional call can then transform that data structure into Python objects without that transformation imposing a performance penalty.

Batch query mode is particularly useful when an algorithm will perform computation on the other side of a hard boundary. When using a GPU, for example, copying a single query point to its memory and then transferring the result back can be a dominating part of the query time, as we will see in Section 4. Similarly, algorithms that use the text-based protocol to communicate with a subprocess benefit enormously from batch query mode; allowing the framework to discount the overhead of string parsing and unparsing reduces the overhead virtually to zero.

3.6 Results and metrics

For each run, we store the full name – including the parameters – of the algorithm instance, the time it took to build its index data structure, and the results of every query: the near neighbours returned by the algorithm, the time it took to find these, and their distances from the query point, along with any additional information the implementation might have exposed. (To avoid affecting the timing of algorithms that do not indicate the distance of a result, the experiment loop independently re-computes distance values after the query has otherwise finished.)

The results of each run are stored in a separate HDF5 file in a directory hierarchy that encodes part of the framework's configuration. Keeping runs in separate files makes them easy to enumerate

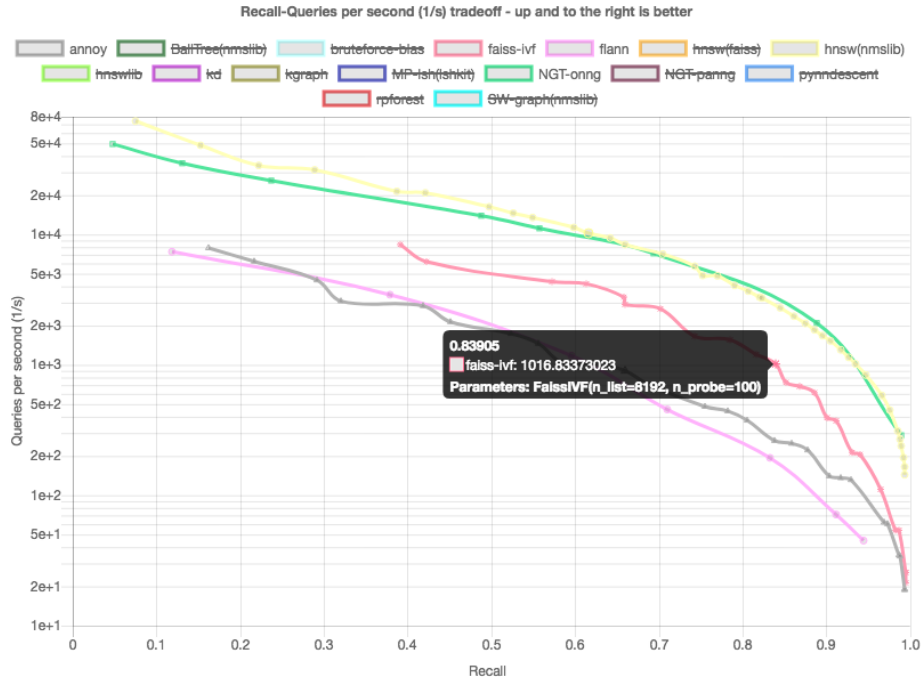

Figure 3 Interactive plot screen from framework's website. Plot shows "Queries per second" (y-axis, log-scaled) against "Recall" (x-axis). Highlighted data point corresponds to a run of FAISS-IVF with parameters as depicted, giving about 1017 queries per second for a recall of about 0.84. The legend shows all available algorithm results; for the sake of presentation, results for the algorithms with struck-out names are not included in the plot.

and easy to re-run, and individual results – or sets of results – can easily be shared to make results more transparent.

Metric functions are passed the ground truth and the results for a particular run; they can then compute their result however they see fit. Adding a new quality metric is a matter of writing a short Python function and adding it to an internal data structure; the plotting scripts query this data structure and will automatically support the new metric.

3.7 Frontend

ANN-Benchmarks provides two options to evaluate the results of the experiments: a script to generate individual plots using Python's matplotlib, and another to generate a website that summarizes the results and provides interactive plots with the option to export the plot as LaTeX code using pgfplots. See Figure 3 for an example. Plots depict the Pareto frontier over all runs of an algorithm; this gives an immediate impression of the algorithm's general characteristics, at the cost of concealing some of the detail. The scripts can also produce scatter plots when more detail is desired.

As batch mode goes to greater lengths to reduce overhead than the normal query mode and exposes more of the system's resources to the implementation being tested, results obtained in batch mode are always presented separately by the evaluation scripts to make the comparisons fairer.

Principle	Algorithms
k-NN graph	`KGraph (KG)` [?], `SWGraph (SWG)` [?, ?], `HNSW` [?, ?], `PyNNDescent (NND)` [?], `PANNG` [?, ?], `ONNG` [?, ?]
tree-based	*FLANN* [?], *BallTree (BT)* [?], `Annoy (A)` [?], `RPForest (RPF)` [?], `MRPT` [?]
LSH	`FALCONN (FAL)` [?], *MPLSH* [?, ?]
other	`Multi-Index Hashing (MIH)` [?] (exact Hamming search), `FAISS-IVF (FAI)` [?] (inverted file)

Table 2 Overview of tested algorithms (abbr. in parentheses). Implementations in *italics* have "recall" as quality measure provided as an input parameter.

Dataset	Data/Query Points	Dimensionality	LID	Metric
SIFT	1 000 000 / 10 000	128	21.9	Euclidean
GIST	1 000 000 / 10 000	960	48.0	Euclidean
GLOVE	1 183 514 / 10 000	100	18.0	Angular/Cosine
NYTimes	234 791 / 10 000	256	18.8	Euclidean
Rand-Euclidean	1 000 000 / 10 000	128	6.8	Angular/Cosine
SIFT-Hamming	1 000 000 / 1 000	256	12.8	Hamming
Word2Bits	399 000 / 1 000	800	24.7	Hamming

Table 3 Datasets under consideration with their local intrinsic dimensionality (LID) computed by MLE [?] from the 100-NN of the queries.

4 Evaluation

In this section we present a short evaluation of our findings from running benchmarks in the benchmarking framework. After a discussion of the evaluated implementations and datasets, we present four questions that we answer using the framework. Subsequently, we discuss the answers to these questions, present some observations regarding the build time of indexes and their ability to answer batched queries, and summarise our findings.

Experimental setup. All experiments were run in Docker containers on *Amazon EC2 c5.4xlarge* instances that are equipped with Intel Xeon Platinum 8124M CPU (16 cores available, 3.00 GHz, 25.0MB Cache) and 32GB of RAM running Amazon Linux. Every experiment was repeated multiple times to verify that performance was reliable, and the results were compared with those obtained on a 4-core Intel Core i7-4790 clocked at 3.6 GHz with 32GB RAM. While the latter was a little faster, the relative order of algorithms remained stable. For each parameter setting and dataset, the algorithm was given five hours to build the index and answer the queries.

Tested Algorithms. Table 2 summarizes the algorithms that are used in the evaluation; see the references provided for details. The framework has support for more implementations and many of these were included in the experiments, but they turned out to be either non-competitive or too similar to other implementations.[1] The scripts that set up the framework automatically fetch the most current version found in each algorithm's repository.

[1] For example, the framework contains three different implementations of `HNSW`: the original one from NMSlib, a standalone variant inspired by that one, and an implementation in `FAISS` that is again inspired by the implementation in NMSlib. The first two implementations perform almost indistinguishably, while the implementation provided in `FAISS` was a bit slower. For the sake of brevity, we also omit the two random projection forest-based methods `RPForest` and `MRPT` since they were always slower than `Annoy`.

In general, the implementations under evaluation can be separated into three main algorithmic principles: graph-based, tree-based, and hashing-based algorithms. *Graph-based algorithms* build a graph in which vertices are the points in the dataset and edges connect vertices that are true nearest neighbors of each other, forming the so-called k-NN graph. Given a query point, close neighbors are found by traversing the graph in a greedy, implementation-specific fashion [**?**, **?**, **?**, **?**, **?**]. *Tree-based algorithms* use a collection of trees as their data structure. In these trees, each node splits the dataset into subsets that are then processed in the children of the node. If the dataset associated with a node is small enough, it is directly stored in the node which is then a leaf in the tree. For example, `Annoy` [**?**] and `RPForest` [**?**] choose in each node a random hyperplane to split the dataset. Given a query point, the collection of trees is traversed to obtain a set of candidate points from which the closest to the query are returned. *Hashing-based algorithms* apply hash functions such as locality-sensitive hashing [**?**] to map data points to hash values. At query time, the query point is hashed and keys colliding with it, or not too far from it using the multi-probe approach [**?**], are retrieved. Among them, those closest to the query point are returned. Different implementations are mainly distinguished by their choice of underlying locality-sensitive hash function.

Datasets. The datasets used in this evaluation are summarized in Table 3. More information on these datasets, as well as results for other datasets, can be found on the framework's website. The NYTimes dataset was generated by building tf-idf descriptors from the bag-of-words version, and embedding them into a lower dimensional space using the Johnson-Lindenstrauss Transform [**?**]. The Hamming space version of SIFT was generated by applying Spherical Hashing [**?**] using the implementation provided by the authors of [**?**]. The dataset Word2Bits comes from the quantized word vector approach described in [**?**] using the top-400 000 words in the English Wikipedia from 2017.

The dataset Rand-Euclidean is generated as follows: Assume that we want to generate a dataset with n data points, n' query points, and are interested in finding the k nearest neighbors for each query point. For an even dimension d, we generate $n - k \cdot n'$ data points of the form $(v, \mathbf{0})$, where v is a random unit length vector of dimension $d/2$, and $\mathbf{0}$ is the vector containing $d/2$ 0 entries. We call the first $d/2$ components the *first part* and the following $d/2$ components the *second part* of the vector. From these points, we randomly pick n' points $(v_1, \ldots, v_{n'})$. For each point v_i, we replace its second part with a random vector of length $1/\sqrt{2}$. The resulting point is the query point q_i. For each q_i, we insert k random points at varying distance increasing from 0.1 to 0.5 to q_i into the original dataset. The idea behind such a dataset is that the vast majority of the dataset looks like a random dataset with little structure for the algorithm to exploit, while each query point has k neighbors that are with high probability well separated from the rest of the data points. This means that the queries are easy to answer locally, but they should be difficult to answer if the algorithm wants to exploit a global structure.

Parameters of Algorithms. Most algorithms do not allow the user to explicitly specify a quality target—in fact, only three implementations from Table 2 provide "recall" as an input parameter. We used our framework to test many parameter settings at once. The detailed settings tested for each algorithm can be found on the framework's website.

Status of FALCONN. While preparing this full version, we noticed that the performance of FALCONN has drastically decreased in recent versions. We communicated this to the authors of [**?**], who are, as of publication time, still working on a fix. For reference, we include the results from the conference version of this paper verbatim in Figure 4, but we do not discuss results related to FALCONN.

4.1 Objectives of the Experiments

We used the benchmarking framework to find answers to the following questions:

(Q1) Performance. Given a dataset, a quality measure and a number k of nearest neighbors to return, how do algorithms compare to each other with respect to different performance measures, such as query time or index size?
(Q2) Robustness. Given an algorithm $\mathcal{A}$, how is its performance and result quality influenced by the dataset and the number of returned neighbors?
(Q3) Approximation. Given a dataset, a number k of nearest neighbors to return, and an algorithm $\mathcal{A}$, how does its performance improve when the returned neighbors can be an approximation? Is the effect comparable for different algorithms?
(Q4) Embeddings. Equipped with a framework with many different datasets and distance metrics, we can try interesting combinations. How do algorithms targeting Euclidean space or Cosine similarity perform in, say, Hamming space? How does replacing the internals of an algorithm with Hamming space related techniques improve its performance?

4.2 Discussion

(Q1) Performance. Figure 4 shows the relationship between an algorithm's achieved recall and the number of queries it can answer per second (its QPS) on the two datasets GLOVE (Cosine similarity) and SIFT (Euclidean distance) for 10- and 100-nearest neighbor queries.

For GLOVE, we observe that the graph-based algorithms clearly outperform the tree-based approaches. It is noteworthy that all implementations, except FLANN, achieve close to perfect recall. Over all recall values, HNSW and ONNG are fastest. However, at high recall values they are closely matched by KGraph. Next comes FAISS-IVF, only losing to the graph-based approaches at very high recall values. For 100 nearest neighbors, the picture is very similar. We note, however, that most graph-based indexes were not able to build indexes for nearly perfect recall values within the five-hour time limit.

On SIFT, all tested algorithms can achieve close to perfect recall. Again, the graph-based algorithms are fastest; they are followed by Annoy and FAISS-IVF. FLANN and BallTree are at the end. In particular, FLANN was not able to finish its auto-tuning for high recall values within the five-hour time limit.

Very few of these algorithms can tune themselves to produce a particular recall value. In particular, almost all of the fastest algorithms on the GLOVE dataset expose many parameters, leaving the user to find the combination that works best. The KGraph algorithm, on the other hand, uses only a single parameter, which—even in its "smallest" choice—still gives high recall on GLOVE and SIFT. FLANN manages to tune itself for a particular recall value well. However, at high recall values, the tuning does not complete within the time limit, especially with 100-NN.

Figure 5 relates an algorithm's performance to its index size. (Note that here down and to the right is better.) High recall can be achieved with small indexes by probing many points; however, this probing is expensive, and so the QPS drops dramatically. To reflect this performance cost, we scale the size of the index by the QPS it achieves for a particular run. This reveals that, on SIFT, most implementations perform similarly under this metric. HNSW and ONNG are best (due to the QPS they achieve), but most of the other algorithm achieve similar cost. In particular, FAISS-IVF and FLANN do well. NND, Annoy, and BallTree achieve their QPS at the cost of relatively large indexes, reflected in a rather large gap between them and their competition. On GLOVE, we see a much wider spread of index size performance. Here, FAISS-IVF and HNSW perform nearly indistinguishably. Next follow the other graph-based algorithms, with FLANN among them. Again, Annoy and BallTree perform worst in this measure.
(Q2) Robustness. Figure 6 plots recall against QPS on the dataset Rand-Euclidean. Recall from our earlier discussion of datasets that this dataset contains easy queries, but requires an algorithm to exploit the local structure instead of some global structure of the data structure, cf. **Datasets**. We

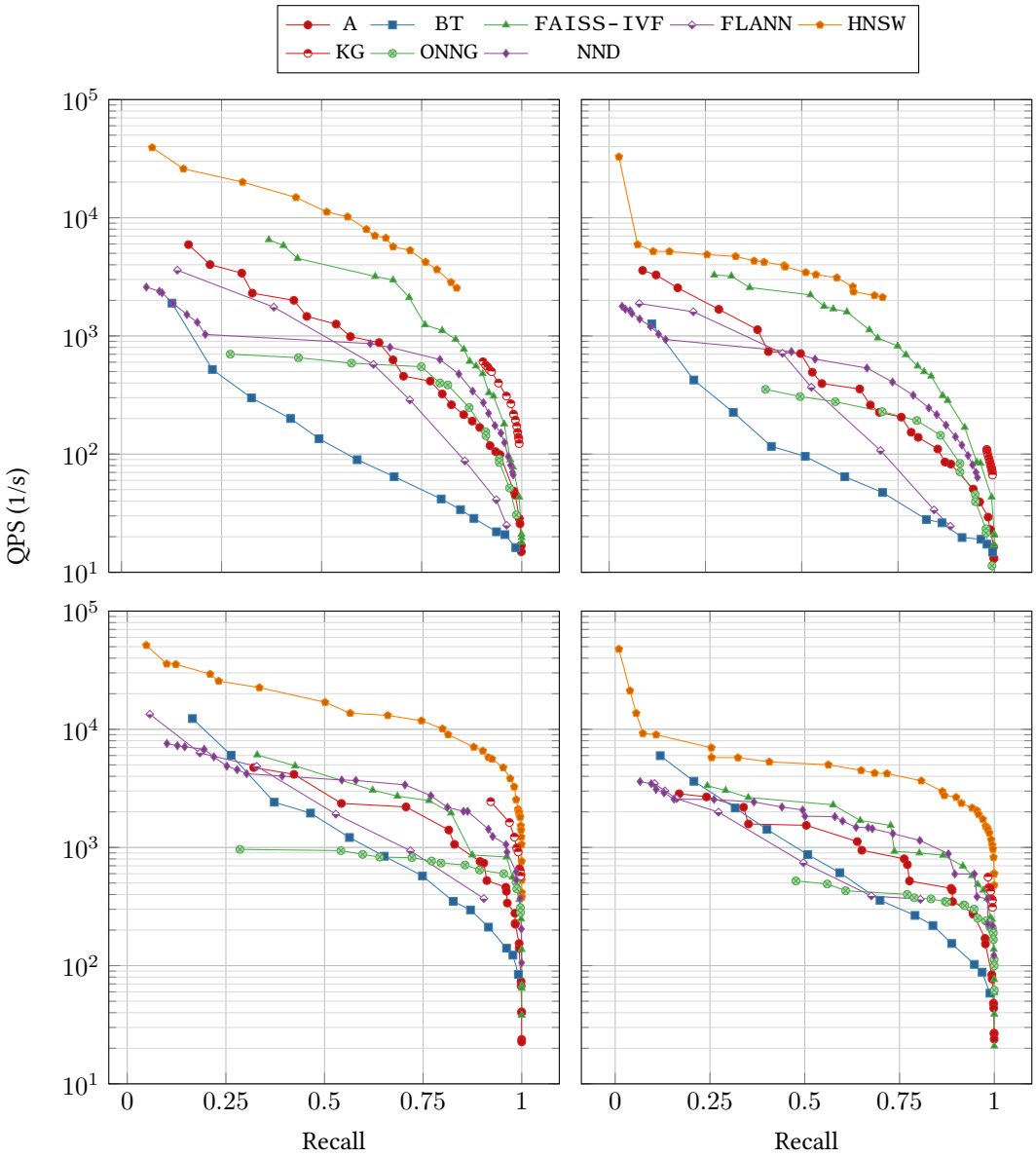

Figure 4 Recall-QPS (1/s) tradeoff - up and to the right is better. Top: GLOVE, bottom: SIFT; left: 10-NN, right: 100-NN.

see very different behavior than before: there is a large difference between different graph-based approaches. While ONNG, KGraph, NND can solve the task easily with high QPS, both HNSW and SWG fail in this task. This means that the "small-world" structure of these two methods *hurts* performance on such a dataset. In particular, no tested parameter setting for HNSW achives recall beyond .86. Annoy performs best at exploiting the local structure of the dataset and is the fastest algorithm. The dataset is also easy for FAISS-IVF, which also has very good performance.

Let us turn our focus to how the algorithms perform on a wide variety of datasets. Figure 7 plots recall against QPS for Annoy, FAISS-IVF, and HNSW over a range of datasets. Interestingly, implementations agree on the "difficulty" of a dataset most of the time, i.e., the relative order of performance is the same among the algorithms. Notable exceptions are Rand-Euclidean, which

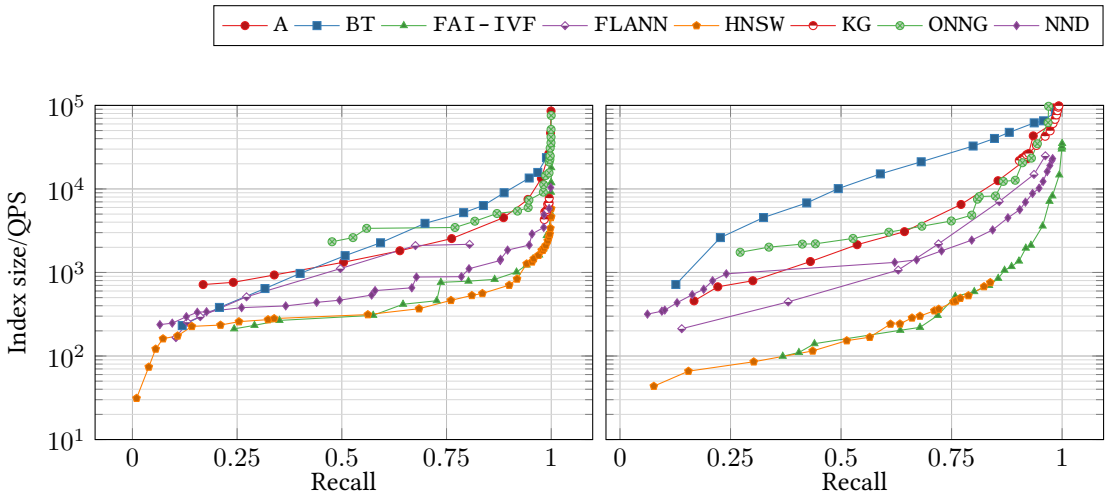

Figure 5 Recall-Index size (kB)/QPS (s) tradeoff - down and to the right is better. Left: SIFT (k=100), right: GLOVE (k=10).

is very easy for Annoy and FAISS-IVF, but difficult for HNSW (see above), and NYTimes, where FAISS-IVF fails to achieve recall above .7 for the tested parameter settings. Although all algorithms take a performance hit for high recall values, HNSW is least affected. On the other hand, HNSW shows the biggest slowdown in answering 100-NN compared to 10-NN queries among the different algorithms.

(Q3) Approximation. Figure 8 relates achieved QPS to the (approximate) recall of an algorithm. The plots show results on the GIST dataset with 100-NN for recall with no approximation and approximation factors of 1.01 and 1.1, respectively. Despite its high dimensionality, all considered algorithms achieve close to perfect recall (left). For an approximation factor of 1.01, i.e., distances to true nearest neighbors are allowed to differ by 1%, all curves move to the right, as expected. Also, the relative difference between the performance of algorithms does not change. However, we see a clear difference between the candidate sets that are returned by algorithms at low recall. For example, the data point for MRPT around .5 recall on the left achieves roughly .6 recall as a 1.01 approximation, which means that roughly 10 new candidates are considered true approximate nearest neighbors. On the other hand, HSNW, FAISS-IVF, and Annoy improve by around 25 candidates being counted as approximate nearest neighbors. We see that allowing a slack of 10% in the distance renders the queries too simple: almost all algorithms achieve near-perfect recall for all of their parameter choices. Interestingly, Annoy becomes the second-fastest algorithm for 1.1 approximation. This means that its candidates at very low recall values were a bit better than the ones obtained by its competitors.

(Q4) Embeddings. Figure 9 shows a comparison between selected algorithms on the binary version of SIFT and a version of the Wikipedia dataset generated by Word2Bits, which is an embedding of word2vec vectors [?] into binary vectors. The performance plot for Annoy in the original Euclidean-space version of SIFT is also shown.

On SIFT, algorithms perform much faster in the embedded Hamming space version compared to the original Euclidean-space version (see Figure 4), which indicates that the queries are easier to answer in the embedded space. (Note here that the dimensionality is actually twice as large.) Multi-index hashing [?], an exact algorithm for Hamming space, shows good performance on SIFT with around 460 QPS.

We created a Hamming space-aware version of Annoy, using popcount for distance computa-

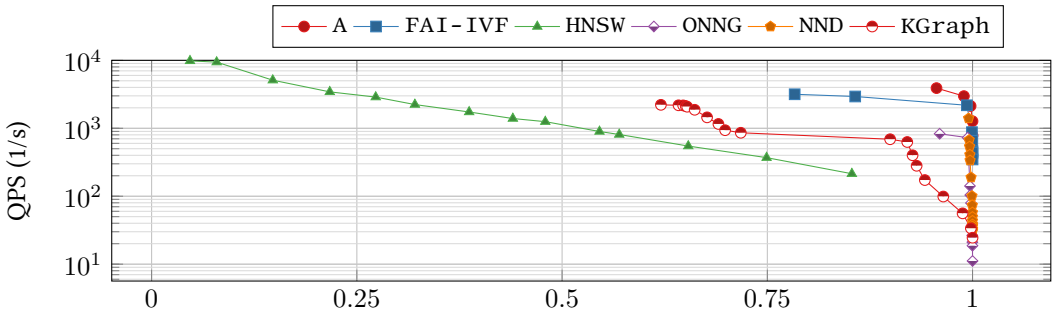

Figure 6 Recall-QPS (1/s) tradeoff - up and to the right is better; Rand-Euclidean with 10-NN.

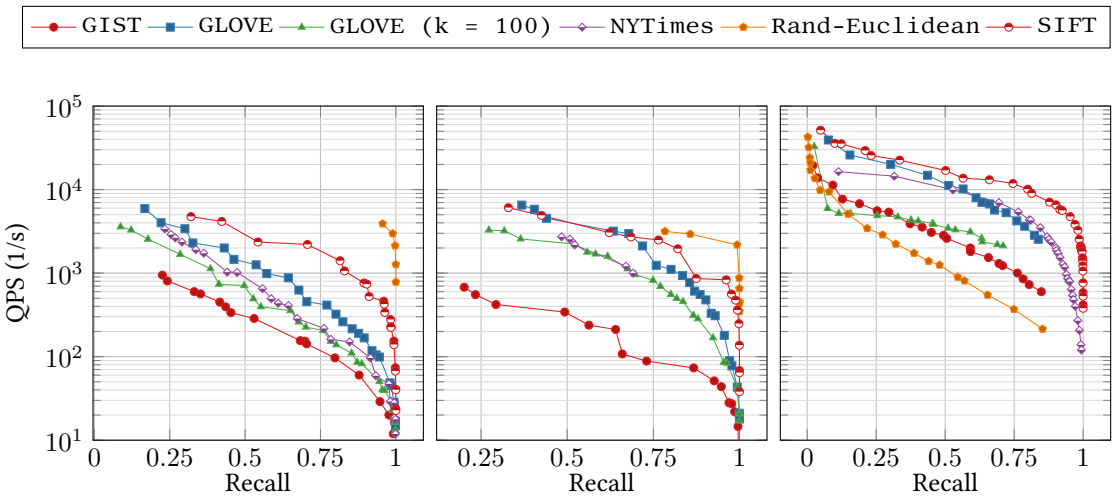

Figure 7 Recall-QPS (1/s) tradeoff - up and to the right is better, 10-nearest neighbors unless otherwise stated, left: Annoy, middle: FAISS-IVF, right: HNSW.

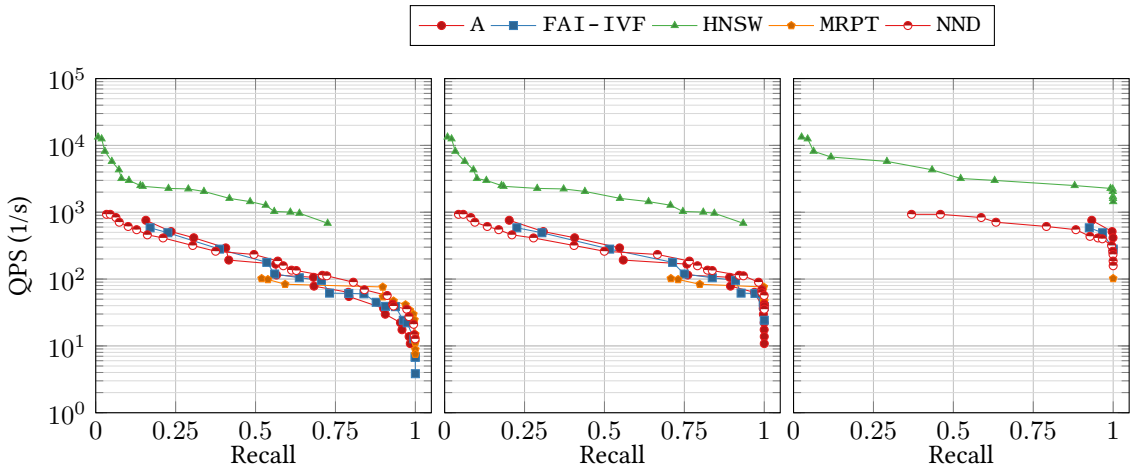

Figure 8 (Approximate) Recall-QPS (1/s) tradeoff - up and to the right is better, GIST dataset, 100-NN; left: $\varepsilon = 0$, middle: $\varepsilon = 0.01$, right: $\varepsilon = 0.1$.

tions, and sampling single bits (as in Bitsampling LSH [?]) instead of choosing hyperplanes. This version is two to three times faster on SIFT until high recall, where the Hamming space version and

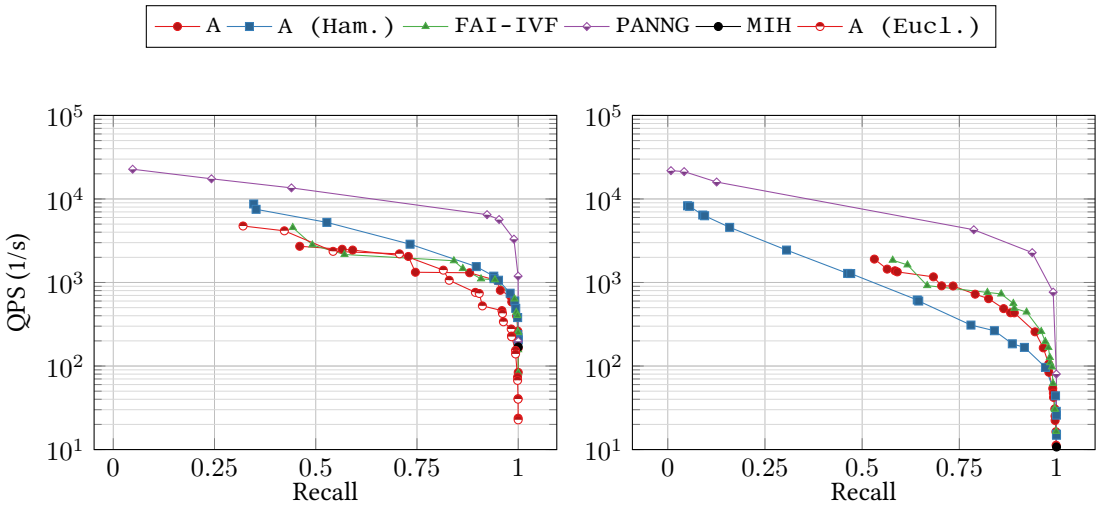

Figure 9 Recall-QPS (1/s) tradeoff - up and to the right is better, 10-nearest neighbors, left: SIFT-Hamming, right: Word2bits. The following versions of Annoy are shown in the plot: A, standard Annoy that uses Euclidean distance as its distance metric; A (Ham.), Annoy with node splitting inspired by Bitsampling LSH and tuned to Hamming space; and A (Eucl.), the run of Annoy on SIFT from Figure 4 (bottom left).

the Euclidean space version converge in running time. On the 800-dimensional Word2Bits dataset the opposite is true and the original version of Annoy is faster than the dedicated Hamming space approach. This means that the original data-dependent node splitting in Annoy adapts better to the query structure than the node splitting by data-independent Bitsampling for this dataset. The dataset seems to be hard in general: MIH achieves only around 20 QPS on Word2Bits. We remark that setting the parameters for MIH correctly is crucial; even though the recall will always be 1, different parameter settings can give wildly different QPS values.

The embedding into Hamming space does have some consistent benefits that we do not show here. Hamming space-aware algorithms should always have smaller index sizes, for example, due to the compactness of bit vectors.

4.3 Index build time remarks

Figure 10 compares different implementations with respect to the time it takes to build the index. We see a huge difference in the index building time among implementations, ranging from FAISS-IVF (around 2 seconds to build the index) to HNSW (almost 5 hours). In general, building the nearest neighbor graph and building a tree data structure takes considerably longer than the inverted file approach taken by FAISS-IVF. Shorter build times make it much quicker to search for the best parameter choices for a dataset. Although all indexes achieve recall of at least 0.9, we did not normalize by the queries per second as in Figure 5. For example, HNSW also achieves its highest QPS with these indexes, but FAISS-IVF needs a larger index to achieve the performance from Figure 4 (which takes around 13 seconds to build). As an aside, FAISS's implementation of HNSW is much faster than the original here, building an index that achieved recall .9 in only 1 700 seconds.

Another perspective on build time and query time is given by Figure 11. There, we plot queries per second *times* the time it took to build the index, which is the build time divided by the average query time. This gives an amortization point: *How many queries must be performed to make it worth building the index structure?* The plot shows that the amortization point decreases for all methods as the recall increases. The differences in building time discussed in Figure 10 translate to very

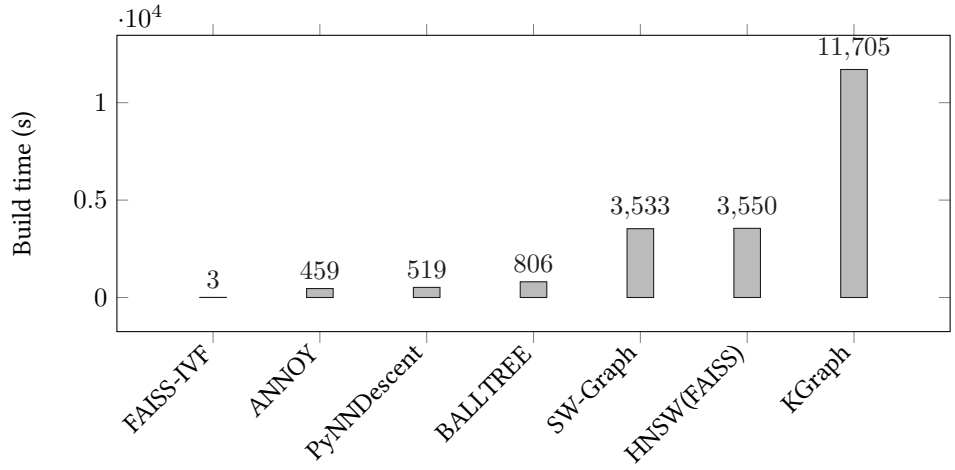

Figure 10 Index build time in seconds for dataset GLOVE. The plot shows the minimum build time for an index that achieved recall of at least 0.9 for 10-NN.

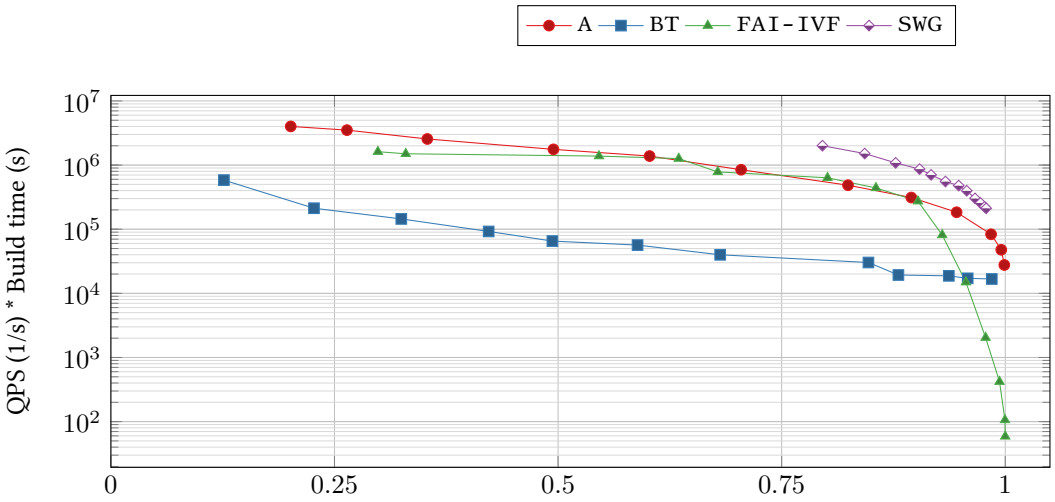

Figure 11 Tradeoff between recall and the number of queries before build time is amortized.

different curves. For example, FAISS-IVF amortizes the time spent building the index after around 100 000 queries for recall 0.9, or 1 000 000 queries at recall 0.75. By comparison, HNSW needs around a factor of 100 more queries to amortize for its longer build times: around 10 million at recall 0.9 and 100 million at recall 0.75.

We remark that it makes little sense to optimize for this cost measure; it is merely useful when deciding whether or not a particular use case justifies building a certain index.

4.4 Batched Queries

We turn our focus to batched queries. In this setting, each algorithm is given the whole set of query points at once and has to return closest neighbors for each point. This allows for several optimizations: in a GPU setting, for example, copying query points to, and results from, the GPU's memory is expensive, and being able to copy everything at once drastically reduces this overhead.

The following experiments have been carried out on an Intel Xeon CPU E5-1650 v3 @ 3.50GHz

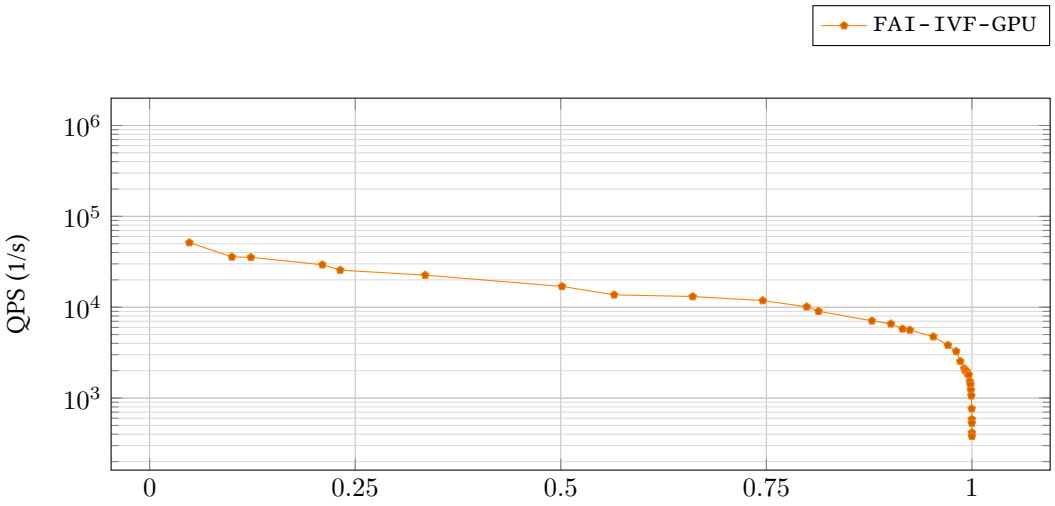

Figure 12 Recall-QPS (1/s) tradeoff - up and to the right is better. Algorithms running batched queries on SIFT with 10-NN. The plot shows a comparison between FAISS' IVF index running on a CPU and a GPU, FAISS' brute force index on the GPU, and HNSW from NMSlib running in batched (B) and non-batched mode (NB).

with 6 physical cores, 15MB L3 Cache, 64 GB RAM, and equipped with an NVIDIA Titan XP GPU.

Figure 12 reports on our results with regard to algorithms in batch mode. FAISS' inverted file index on the GPU is by far the fastest index, answering around 655 000 queries per second for .7 recall, and 61 000 queries per second for recall .99. It is around 20 to 30 times faster than the respective data structure running on the CPU. Comparing HNSW's performance with batched queries against non-batched queries shows a speedup by a factor of roughly 3 at .5 recall, and a factor of nearly 5 at recall .99 in favor of batched queries. It is particularly interesting to see that even the simplest GPU-driven approach, FAISS' brute-force variant, can handle nearly 25,000 queries per second.

4.5 Summary

Which method to choose? From the evaluation, we see that graph-based algorithms provide by far the highest number of queries per second on most of the datasets. HNSW and ONNG are often the fastest algorithms, with ONNG being more robust if there is no global structure in the dataset, according to the experiments presented here. The downside of graph-based approaches is the high preprocessing time needed to build their data structures. This could mean that they might not be the preferred choice if the dataset changes regularly. When it comes to small and quick-to-build index data structures, FAISS' inverted file index provides a suitable choice that still gives good performance in answering queries, as discussed in Section 4.3.

How well do these results generalize? In our experiments, we observed that, for the standard datasets under consideration, algorithms usually agree on

(i) the order in how well they perform on datasets, i.e., if algorithm A answers queries on dataset X faster than on dataset Y, then so will algorithm B; and
(ii) their relative order to each other, i.e., if algorithm A is faster than algorithm B on dataset X, this will most likely be the order for dataset Y.

There exist exceptions from this rule, e.g., for the dataset Rand-Euclidean described above.

How robust are parameter choices? With very few exceptions (see Table 2), users often have to set many parameters themselves. Our framework allows them to choose the best parameter choice

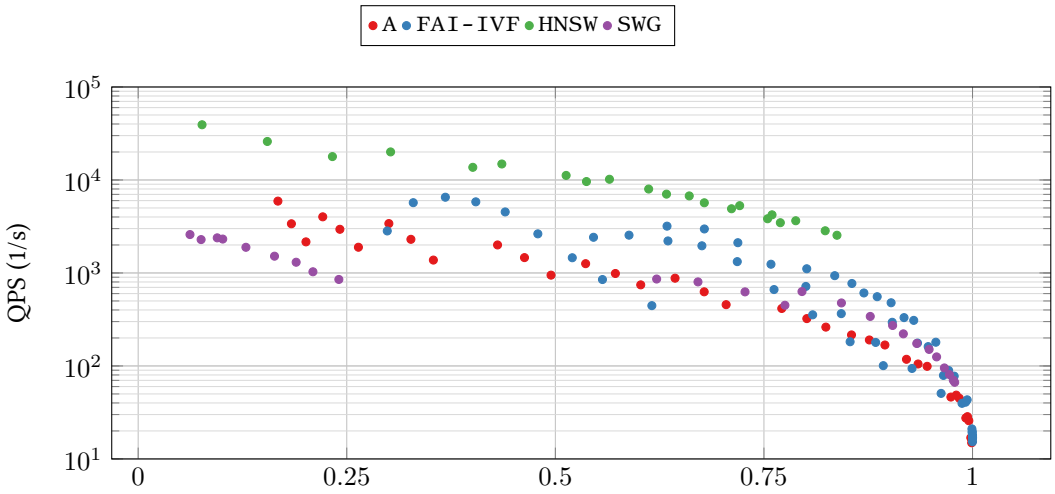

Figure 13 Scatter plot of Recall-QPS (1/s) tradeoff - up and to the right is better on GLOVE with 10-NN.

by exploring the interactive plots that contain the parameter choices that achieve certain quality guarantees.

In general, the *build parameters* can be used to estimate the size of the index[2], while the *query parameters* suggest the amount of effort that is put into searching the index.

We will concentrate for a moment on Figure 13. This figure presents a scatter plot of selected algorithms for GLOVE on 10-NN, cf. the Pareto curve in Figure 4 (in the top left). Each algorithm has a very distinctive parameter space plot.

For HNSW, almost all data points lie on the Pareto curve. This means that the different build parameters blend seamlessly into each other. For Annoy, we see that data points are grouped into clusters of three points each, which represent exactly the three different index choices that are built by the algorithm. For low recall, there is a big performance penalty for choosing a too large index; at high recall, the different build parameters blend almost into each other. For SW-Graph, we see two groups of data points, representing two different index choices. We see that with the index choice to the left, only very low recall is achieved on the dataset. Extrapolating from the curve, choosing query parameters that would explore a large part of the index will probably lead to low QPS. No clear picture is visible for FAISS-IVF from the plot. This is chiefly because we test many different build parameters – recall that the index building time is very low. Each build parameter has its very own curve with respect to the different query parameters.

As a rule of thumb, when aiming for high recall values, a larger index performs better than a smaller index and is more robust to the choice of query parameters.

How do these results generalize to lower-dimensional datasets? While our focus has been on high-dimensional datasets, one might reasonably wonder to what extent the observations made above are true for lower-dimensional datasets. To this end, we embedded the NYTimes dataset so that each vector is in $\mathbb{R}^6$. The relative performance of the implementations discussed earlier is nearly unaffected by this change, i.e., graph-based approaches still provide a better QPS/recall tradeoff than other approaches. However, the exact KD-tree implementation provided with Python's `sklearn` – which performs worse than a linear scan on all of the other datasets in our evaluation –

[2] As an example, the developers of FAISS provide a detailed description of the space usage of their indexes at https://github.com/facebookresearch/faiss/wiki/Faiss-indexes.

becomes very competitive, achieving around 7000 QPS – lower than HNSW (40000 QPS with recall .9984) and FAI-IVF (10000 QPS at recall .9983), but faster, for example, than PyNNDescent, which achieves around 2000 QPS at recall .999. This suggests that exact algorithms are worth considering when working with lower-dimensional datasets.

5 Conclusion & Further Work

We introduced ANN-Benchmarks, an automated benchmarking system for approximate nearest-neighbor algorithms. We described the system and used it to evaluate existing algorithms. Our evaluation showed that well-engineered solutions for Euclidean and Cosine distance exist, and many techniques allow for fast nearest-neighbor search algorithms. At the moment, graph-based approaches such as HNSW or ONNG generally outperform the other approaches for very high recalls, although they can be tripped up by some kinds of dataset. Index building for graph-based approaches takes a long time for datasets with difficult queries.

In future, we aim to add support for other metrics and quality measures, such as positional errors [?]. Preliminary support exists for set similarity under Jaccard distance, but algorithm implementations are missing. Additionally, similarity joins are an interesting variation of the problem worth benchmarking [?]. We remark that the data we store with each algorithm run allows for more analysis beyond looking at average query times. One could, for example, already look at the variance of running times between algorithms, which could yield insights when comparing different approaches. We also intend to simplify and further automate the process of re-running benchmarks when new versions of algorithm implementations appear.

As a general direction for future work, we remark that none of the most performant implementations are easy to use. From a user perspective, the internal parameters of the data structure would ideally be invisible; an algorithm should be able to tune itself for the dataset at hand, given just a handful of quality-related parameters (such as the desired recall or the index size).

In the future, we plan to include a benchmarking mode for investigating this. A new tuning step will be added to the framework, letting implementations examine a small part of the dataset and to tune themselves for some given quality parameters before training begins. Implementors of algorithms would be able to test their auto-tuning techniques easily with such a benchmarking mode.

Another general direction for future work is to get a better understanding which properties of a dataset make it easy or difficult for a specific algorithm. As shown in the evaluation, for many of the real-world datasets we get a homogeneous picture of how well algorithms perform against each other. On the other hand, we have given an example for a random dataset where implementations behave very differently. Which properties of a dataset make it simple or difficult for a specific algorithmic approach? For example, for graph-based algorithms there has been very little research except [?] on the theoretical guarantees that they achieve.

Finally, answering batched queries, in particular on the GPU, is an interesting area for future work. There exist both novel LSH-based implementations [?] of nearest-neighbor algorithms and ideas on how to parallelize queries beyond running each query individually [?]. In particular, for a batch of queries an algorithm should exploit that individual queries might be close to each other.

Acknowledgements: We thank both the reviewers of the earlier conference submission and of this journal submission for their careful comments that allowed us to improve the paper. The first and third authors thank all members of the algorithm group at the IT University of Copenhagen for fruitful discussions. In particular, we thank Rasmus Pagh for the suggestion of the random dataset. This work was supported by a GPU donation from NVIDIA.