OAK

Cedric Sirianni, Mithi Jethwa, Lachlan Kermode

ACM Reference Format:

Cedric Sirianni, Mithi Jethwa, Lachlan Kermode. 2024. OAK. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ABSTRACT

Vector databases are growing in popularity as they become widely used in similarity search and RAG (Retrieval Augmented Generation) systems as part of ML workloads. Recent work in ACORN helps improve the feasibility of *hybrid search* by providing a performant and predicate-agnostic index built on Hierarchical Navigable Small Worlds (HNSW), a state-of-the-art graph based index for approximate nearest neighbor search (ANNS). This paper presents OAK, a system that improves upon ACORN's performance for certain query loads by routing to subgraph indexes when there are gains to be had. To evaluate OAK, we compare OAK to ACORN … `TODO`. We show that OAK achieves improved performance … `TODO`. Our code is available on Github [7].

INTRODUCTION

In recent years, vector similarity search has become a staple method in the arsenal of engineering tools for ML applications. Recommendation algorithms at companies like Netflix, Spotify, and TikTok use vector embeddings to represent user profiles and platform content. In these applications, approximate nearest neighbor search (ANNS) is the method used for performant semantic similarity search. Because machine learning inherently clusters/groups data, ANNS is a way to coalesce inherently unstructured data for specific operations, making it an essential primitive in big data operations.

This growth is in no small part due to the success of machine-learning vectorization methods such as BERT [5] which generate embeddings that seem to preserve something semantic about the original document, a result that turns vector similarity search into a powerful heuristic for *semantic* search. The generality and potential of vector similarity search in pools of *unstructured* data is limited, however, by the relative complexity of its interoperation with more traditional *structured* search methods in databases, whereby databases can deliver better performance and correctness characteristics thanks to decades of research in SQL and SQL-like query optimization.

One increasingly important use-case for ANNS, for example, is retrieval-augmented generation (RAG), which helps improve the accuracy and relevance of LLM outputs by including additional vector embeddings in user prompts. In RAG it is often desirable to perform ANNS with predicates that help to narrow down the relevance of results in this task, for example finding similar prompts from the same user, or from users in the same organization. Creating separate vector databases for predicates manually and maintaining routing strategies among them is unwieldy and suboptimal in terms of space, as many there may be significant overlap between many of the indexes, and deciding which predicate subgraphs to build manually requires engineering effort and effective observability instrumentation on the part of the vector database client.

Executing **hybrid search**, i.e. vector similarity search results are additionally constrained by unbounded and arbitrary predicates over the set of all possible vectors in a vector database, is therefore an important problem in vector databases, as it relieves the database client of managing this complexity. The performance profile of ANNS becomes more complex when introducing such predicate filtering, however, as will be discussed in Section **??**. One recent approach, ACORN [9], proposes an architecture for a *predicate-agnostic* ANNS index that addresses some of these performance complexities.

In this paper, we present **OAK** (Opportunistic ANNS K-Subgraphs), a system that applies ACORN's principle of predicate-agnostic hybrid vector similarity search in a system that offers better performance for certain query loads by maintaining indexes of predicate subgraphs in addition to a base index and intelligently routing to these indexes when there is significant overlap between such an index and a query predicate.

RESEARCH PROBLEM/MOTIVATION

We now discuss the existing predicate filtering strategies.

Pre-filtering first finds all vectors that satisfy a given predicate and then performs a similarity search on the remaining vectors. This approach performs poorly when using medium to high selectivity predicates on large datasets.

Post-filtering first performs a similarity search on the dataset, then filters results that do not match the given predicate. Since vectors with the greatest similarity may not satisfy the predicate, this approach sometimes requires searching

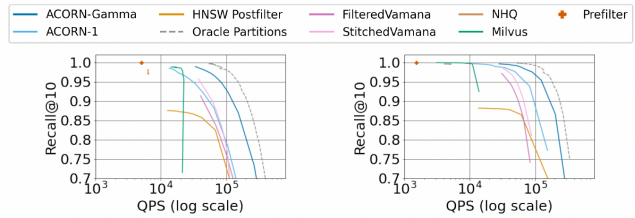
Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

repeatedly with increasingly large search spaces (top-1k, top-10k, etc.), incurring large amounts of overhead.

Specialized indexes such as Filtered-DiskANN [cite:] use predicates during index construction to eliminate the need for pre- or post-filtering. However, these indexes restrict predicate set cardinalities to about 1,000 and only support equality predicates.

Recently, ACORN [9] has proposed a *predicate-agnostic* index which supports unbounded and arbitrary predicates. The results are impressive but still fall short of an *oracle partition index* (Figure 1).



(a) SIFT1M Dataset (b) Paper Dataset

Given some search predicate p and dataset X, an oracle partition index is an index on X_p. An oracle partition index is more performant compared to a base ACORN index because the search space is guaranteed to contain only vectors that satisfy the predicate, and thus no further filtering needs to occur before, during, or after the search itself. In many cases, it is infeasible to construct an oracle index for every possible predicate, and thus a general-purpose index such as ACORN is more desirable, as it performs reasonably well across a diverse array of queries and is much more space efficient.

There are, however, many query distributions could be get significant and meaningful performance improvement if an oracle index were to exist that could be searched rather than an index that also contains vectors that don't match the search predicate. The greater the scale of the database and the volume of queries, the more critical such a performance improvement could be. Consider, for example, queries about football teams during the Superbowl, or about states during election night. In scenarios like this, a general-purpose predicate-agnostic index can be supplemented by carefully chosen supplemental indexes that are constructed so as to effectively serve some frequently occurring search predicates. Intelligently routing to supplemental indexes in such cases is the goal of OAK.

BACKGROUND AND RELATED WORK

Distributed Vector Databases

Our project began as an attempt to build a distributed vector database. Though OAK is not inherently distributed, our learnings from this literature review greatly impacted our decision to build OAK and influenced its design. (We also outline OAK's distributed horizons in Section **??**.) This literature is relevant as it introduces the idea of routing each query between a series of choices, and selecting and/or aggregating results.

Replication. Replication is the simplest way of distributing a vector database. Though it can improve fault tolerance and throughput (as replication can in a regular database), it does not have any benefit in terms of **scalability**, i.e. constructing a database that can store a greater number of vectors than on a single node.

Random Partitioning. When an index does not fit in RAM, it can be partitioned across multiple nodes. Many commerical services [1, 3] support this functionality through random partioning.

The general approach is as follows:

(1) Split the entire database into random, equally-sized partitions.
(2) For each data node, train an index on one partition.

This design addresses the memory constraint incurred by large datasets but is not performant. The router must dispatch every query to every node, potentially scanning many unnecessary vectors during the search process [11]. Thus random partitioning does not offer tangible benefit in terms of throughput.

Balanced Partitioning. Random partitioning can be improved by searching only nodes relevant to the query vector. Balanced partitioning implements this idea by partitioning vectors based on similarity.

The approach is analogous to sharding in a RDBMS or KV Store. Concretely, shards are computed by partitioning vertices into roughly equal size sets such that the number of edges which connect different sets is approximately minimized. A shardmaster/router can then dispatch queries to only a subset of the total shards, thus improving performance and scalability.

Balanced graph partitioning is a well-studied problem. For example, Pyramid [4] proposes building a much smaller meta-HNSW that captures the structure of the entire dataset, which can then be used to route queries to the correct partition HNSW index(es). More recently, Gottesburen et al. proposed a balanced graph partitioning strategy using k-means clustering with theoretical routing guarantees [6].

Despite these advancements, balanced partitioning is still not supported on platforms such as FAISS or Qdrant. This could be explained by the significantly greater complexity of balanced partitioning compared to random partitioning, which requires substantial engineering effort to implement.

HNSW and ACORN

The underlying data structure that enables vector similarity search in both ACORN and OAK is Hierarchical Navigable Small Worlds (HNSW) [8], an index that makes use of a hierarchical, tree-link structure during search to more quickly

work through the plausible search space while retaining reasonable recall. More specifically, HNSW leverages a *proximity graph* [10], in which two vertices are linked based on proximity. Proximity is usually computed using Euclidean distance, though other similarity metrics exist (e.g. cosine similarity).

ACORN proposes two HNSW variants entitled **ACORN-γ** and **ACORN-1**. ACORN-γ is designed to achieve efficient search performance, and ACORN-1 is designed achieve similar performance while reducing the time to index (TTI) and space footprint.

ACORN-γ modifies the HNSW construction algorithm by introducing *neighbor expansion*, which creates a denser graph. While HNSW collects M approximate nearest neighbors as candidate edges for each node in the index, ACORN-γ collects $M \cdot \gamma$ approximate nearest neighbors as candidate edges per node. The intuition is that given enough redundant nodes, the search space is sufficiently large, even when filtering based on the predicate during search.

This is not always the case, though. If the predicate selectivity falls below a minimum specified threshold, ACORN resorts to pre-filtering and brute force search, favoring recall over performance. This may explain the difference in throughput between ACORN-γ and the opportunistic index in Figure 1.

ACORN-1 performs neighbor expansion during search, not construction. Thus ACORN-1 collects exactly M approximate nearest neighbors as candidates edges per node during construction.

MAIN DESIGN

The central premise of OAK is to route queries with high-frequency predicates to an *opportunistic index* constructed using the same or a significantly similar predicate. When OAK receives a query q with predicate p, sending to an opportunistic index is

(1) potentially more performant (if the base index is larger than the opportunistic index)
(2) potentially less accurate (if the opportunistic index does not contain all vectors that match p).

OAK's query routing strategy leverages these insights to manage the performance-accuracy tradeoff at hand. It is worth mentioning that this tradeoff is inherent to the ANNS problem space as a whole, and that using additional space in order to deliver better latency is the specific goal of ANNS.

In order to make routing decisions, opportunistic indexes are stored as values in a hash map for which the key is a bitmask. Each bitmask/key is constructed over all vectors in the database, so as to provide OAK with an efficient way to consider which vectors exist in available opportunistic indexes.

Query routing. The key insight for OAK's routing strategy is that queries can be sent to an opportunistic subindex even if the query bitmask M_q and the subindex bitmask M_s do not wholly overlap. M_x denotes a bitmask over OAK's root index r (which contains *all* vectors in the database, regardless of predicate).

If M_q is a strict subset of (or is equal to) M_{s_i}, then similarity search on s_i will be strictly preferable to search on r. We use a bitmask's **Hamming weight** (the number of ones in the mask) as a proxy for how much more performant a subindex will be. The smaller the Hamming weight of M_{s_i}, the more performant a search within s_i will be in comparison a search in r. We consider this a reasonable heuristic for OAK's current implementation in part because all indexes (both r and all s_i) are ACORN indexes constructed with the same base parameters.

If M_q is *not* a subset of M_{s_i}, then we consider the **Jaccard similarity** between M_q and M_{s_i} as a proxy for how significant the reduction in recall will be when routing to s_i. If there is significant overlap between M_{s_i} and M_q, and if the number of vectors specified by the query (k) is significantly high, then we consider it more desirable to service the query in s than in r.

Concretely, we consider servicing a query in subindex s_i preferable to the base index r if it satisfies the following condition:

$$M_q \subseteq M_{s_i} \quad \vee \quad \frac{|r|}{|s_i|} \cdot J\left(M_q, M_{s_i}\right) - \text{efRouting} \geq 0$$
$$\text{where} \quad J\left(M_q, M_{s_i}\right) = \frac{|M_q \cap M_{s_i}|}{|M_q \cup M_{s_i}|} \tag{1}$$

Though this routing strategy is yet untested, we present it here to detail a basic approach that could be modified with experimentation. `efRouting` is considered the threshold at which the performance gain $\frac{|r|}{|s_i|}$ is significant enough that the recall degradation $J\left(M_q, M_{s_i}\right)$ is acceptable. (We acknowledge that this routing strategy requires several more phases of conceptual iteration in order to result in meaningful improvement over search in r; we present it here as demonstrative of our thinking.)

IMPLEMENTATION

OAK is built in approximately 1k lines of Rust. We encountered three main engineering challenges, which we now discuss in sections.

Bindings

ACORN is implemented in C++, so writing OAK in Rust required a foreign function interface (FFI). We originally chose bindgen [2] to automatically generate Rust FFI bindings to

ACORN, but realized that the task would require a substantial engineering effort as ACORN's functionality was only implemented in C++ and not exposed to the C API that FAISS [1] (the codebase from which ACORN was forked) provides. We thus chose instead to use cxx [12] to interoperate directly between C++ and Rust, an approach that worked well once we had worked through the new conceptual model for FFI. Our progress implementing the FFI layer was slowed, too, on a number of other counts: we discovered a bug in the ACORN compilation directions that resulted in PR (https://github.com/guestrin-lab/ACORN/pull/7); the FFI wrappers introduced lifetime issues in C++ when dereferencing a unique pointer; and the functionality of the ACORN `search` function had to be largely reverse-engineered due to a lack of clarifying documentation.

Predicate Filtering

Though predicates are represented internally in OAK as bitmasks (`Vec<i8>`), we introduce a `PredicateQuery` struct to allow clients to more concisely express their filters during search. `PredicateQuery` can be used to generate a bitmask over the vectors in OAK by specifying a `PredicateOp` such as `Equals`, and a `PredicateRhs` such as `5`. Though the equality of `i8` values is sufficient for the experiments outlined in this paper, `PredicateQuery` is designed to be extensible for arbitrary predicate operations and right-hand-sides, such as `LessThan` and `Regex`. For each vector, if the predicate is true (e.g. year = 2024), the element in the generated bitmask is set to 1. The ACORN `search` function, by contrast, accepts a `filter_id_map` bit mask that is used to filter vectors that do not satisfy the predicate during search.

SimilaritySearchable

The `SimilaritySearchable` trait is the mechanism through which OAK clients can easily switch between using OAK's base index r, a subindex s_i, or an index dynamically selected by way of the routing strategy described in Section **??** to search for similar vectors. Each of the structs that represent these three entities implement the `SimilaritySearchable` trait.

Structs that implement the trait offer three methods:

`initialize`. Takes one `OakIndexOptions` argument, and prepares the search mechanism.

`search`. Takes the `query_vectors` (multiple are permitted for batch search), an optional `PredicateQuery`, and a `topk` specification. This method relieves the client of the complexity of generating a predicate bitmask.

`search_with_bitmask`. Takes the `query_vectors`, a `Bitmask` (constructed over all available searchable vectors), and a `topk` specification. This is the method that we evaluate in the next section.

EVALUATION

`TODO lachlan and mithi`

FUTURE WORK

OAK has many opportunities for future work.

Dynamic index construction. Right now, OAK constructs indexes only once and before queries are dispatched. In a production system, it may be advantageous to construct indexes while queries are received to increase throughput. The overhead incurred by index construction could be measured with respect to the time TTI, the size of the index, and the compute/memory resources required to construct the index.

Index configuration and type. OAK uses ACORN for opportunistic indexes primarily because writing bindings to additional indexes is unnecessary for a proof-of-concept. The construction parameters for an ACORN index affect its recall-performance tradeoff, however, and it would be interesting to consider whether variations of ACORN index could factor into a more efficient subindex architecture. However, given the bounded and well-defined nature of the opportunistic index type, a *specialized index* may yield better performance. For example, Qdrant [[**https://qdrant.tech/articles/filtrable-hnsw?**]/] has proposed denser HNSW graph by knowledge of the search predicates to add additional edges. While this is ill-suited for ACORN's goal to be predicate-agnostic, the principle idea of opportunistic indexes is *predicate-knowledge*, and thus we can leverage the known predicate to construct a better index.

Distribution. The system design of OAK is also easily transferrable to a distributed context. We could construct and/or host indexes on different nodes, as network communication costs are dominated by the ANNS latency. [cite:] This would help remove the bottleneck of commodity hardware when hosting multiple indexes and enable horizontal scaling and load balancing during bursty workloads.

LOGISTICS

Much of the literature review in the early stages of our project was done collectively. Lachlan and Cedric implemented the FFI interface between OAK and ACORN. Mithi and Lachlan implemented the SIFT dataset preparation scripts (as attributes needed to be randomly added to the vectors there, following the approach in the ACORN paper). Though the concepts and talking points for the presentation were discussed and determined collectively, Cedric was principally responsible for the visual integrity and flair. Mithi implemented the experimentation infrastructure, as well as the graph generation code. Lachlan conceived and implemented OAK's routing logic (and is solely responsible for any senselessness it contains). Mithi and Lachlan worked together to run the experiments on AWS. Cedric drafted and was chief editor for the final paper, though it was collectively edited and written.

BIBLIOGRAPHY

[1] 2024. Facebookresearch/faiss.
[2] 2024. Rust-lang/rust-bindgen.

[3] Distributed Deployment - Qdrant.

[4] Shiyuan Deng, Xiao Yan, K. W. Ng Kelvin, Chenyu Jiang, and James Cheng. 2019. Pyramid: A General Framework for Distributed Similarity Search on Large-scale Datasets. In *2019 IEEE International Conference on Big Data (Big Data)*, December 2019. 1066–1071. https://doi.org/10.1109/BigData47090.2019.9006219

[5] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018). Retrieved from https://arxiv.org/abs/1810.04805

[6] Lars Gottesbüren, Laxman Dhulipala, Rajesh Jayaram, and Jakub Lacki. 2024. Unleashing Graph Partitioning for Large-Scale Nearest Neighbor Search. Retrieved September 28, 2024 from https://arxiv.org/abs/2403.01797

[7] Lachlan Kermode. 2024. Breezykermo/oak.

[8] Yu A. Malkov and Dmitry A. Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[9] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. https://doi.org/10.48550/arXiv.2403.04871

[10] Pinecone. Hierarchical Navigable Small Worlds (HNSW). *Faiss: The Missing Manual*.

[11] Yuxin Sun. 2024. A Distributed System for Large Scale Vector Search. Master's thesis. ETH Zurich. https://doi.org/10.3929/ethz-b-000664643

[12] David Tolnay. 2024. Dtolnay/cxx.