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, 2 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 systems as part of ML workloads. At the same time, applications increasingly leverage mixed-modality data, requiring support for search over *vector data* such as images and text, and *structured data* such as metadata and keywords, simultaneously. Recent work in ACORN helps improve the feasibility of this *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). However, ACORN does not take into consideration predicate access patterns, leaving room for improved performance under certain modal workloads. To address this, we present OAK, a system for creating predicate subgraphs. To evaluate OAK, we compare OAK to ACORN on ….. We show that OAK achieves improved performance … Our code is available at: https://github.com/breezykermo/oak.

INTRODUCTION

In recent years, embedding models have advanced significantly, enabling the use of vector embeddings in a variety of applications. For example, retrieval-augmented generation (RAG) helps improve the accuracy and relevance of LLM outputs by including additional vector embeddings in user prompts. And, recommendation algorithms at companies like Netflix 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.

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

However, efficient ANNS is a challenging problem when considering multi-million or billion scale datasets. Both NeurIPS'21 [[**https://big-ann-benchmarks.com/neurips21.html?**]]and NeurIPS'23 [[**https://big-ann-benchmarks.com/neurips23.html?**]] hosted a competition for billion-scale indexing data structures and search algorithms, showcasing a wide range of solutions that improved search accuracy and efficiency.

ANNS becomes increasingly complex when introducing predicate filtering, i.e., performing on ANNS on a subset of data that matches a given predicate. For example, customers on an e-commerce site may want to search for t-shirts similar to a reference image, while filtering on price. To support such functionality, applications must implement *hybrid search*, i.e., similarity search queries containing a one or more structured predicates. Implementations must be **performant**, retrieving results with high throughput/low latency and also **accurate**, retrieving results that are sufficiently similar to the provided query.

Several strategies exist to address these challenges with varying degrees of success, including pre-filtering, post-filtering, and specialized indices. One approach is particularly promsing: ACORN [cite:], which proposes a *predicate-agnostic* index over unbounded and arbitrary predicates. But there is still room for improvement.

In this paper, we present **OAK** (Opportunistic ANNS K-Subgraphs), a system that combines predicate-agnostic indices with secondary indices to support arbitary/unbounded predicates and high performance for high-frequency query predicates.

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

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

Recently, ACORN [cite:] 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).

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 predicate comparison must occur during the search itself. The reason ACORN provides a *predicate-agnostic* index is because in many systems, it is infeasible to construct an index for every possible predicate, and thus a general-purpose index performs well across a diverse array of queries.

However, access patterns are sometimes not uniformly distributed. Consider, for example, queries about football teams during Superbowl or states during election night. In certain scenarios, a general-purpose predicate-agnostic index can be supplemented by carefully chosen supplemental indices constructed using frequently occurring predicates. This is the goal of OAK.

BACKGROUND AND RELATED WORK

The underlying data structure of both ACORN and OAK is Hierarchical Navigable Small Worlds (HNSW)[[**https://arxiv.org/abs/1603.09320?**]], a hierarchical, tree-link structure where each node of the tree represents a set of vectors. More specifically, it is a *proximity graph*, 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 modifies the HNSW contruction algorithm to use neighbor expansion, creating a denser graph. While HSNW collects M approximate nearest neighbors as candidate edges for each node in the index, ACORN collects $M\gamma$ 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 theshold, 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.

MAIN DESIGN

The central premise of OAK is to route queries with high-frequency predicates to an *opportunistic index* constructed using the same 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) but (2) potentially less accurate (if the opportunistic index does not contain all vectors that match p). We factor this performance-accuracy tradeoff into our query routing strategy.

IMPLEMENTATION

OAK is built in approximately 700 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 chose bindgen [cite:] to automatically generate Rust FFI bindings to ACORN, but the task required a substantial engineering effort. 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 when dereferencing a unique pointer, and the functionality of the ACORN `search` function had to be largely reverse-engineered due to a lack of documentaiton.

Predicate Filtering

To programmatically represent predicates, we introduce the 'PredicateQuery' struct which contains a predicate operator (e.g. equality) and an operand. This 'PredicateQuery' is used to generate a bit mask over the set of vectors and is designed for any arbitrary predicate. For each vector, if the predicate is true (e.g. year == 2024), the corresponding bit mask element is set to 1. The ACORN `search` function accepts a `filter_id_map` bit mask that is used to filter vectors that do not satisfy the predicate during search.

Query Routing

EVALUATION

FUTURE WORK

LOGISTICS

BIBLIOGRAPHY