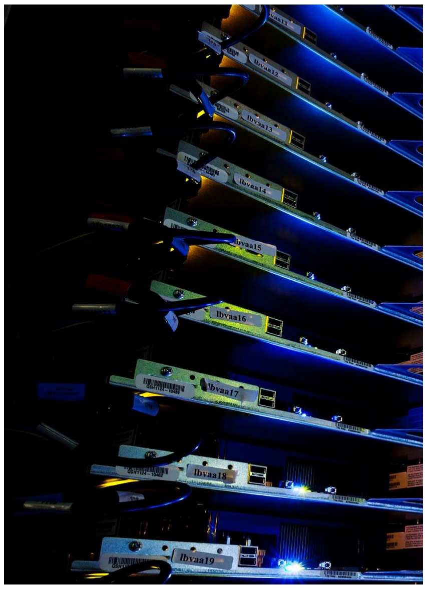
ScaNN for AlloyDB

Google Cloud



Authors

Yannis Papakonstantinou
Distinguished Engineer, Databases

Alan Li
Software Engineer, Databases

Ruiqi Guo
Software Engineer, Google Research

Sanjiv Kumar
Google Fellow

Phil Sun
Software Engineer, Google Research

April 2024

Next gen vector search by bringing 12 years of Google research to databases

Vector embeddings are widely used to represent the semantics of unstructured data such as text, audio, images, and video. Nowadays, relational database users employ **embedding models** to generate embeddings for each piece of unstructured data and then store these embeddings in the database. Consequently, finding data that are semantically related to a user request turns into a vector proximity operation: find the database embeddings that are the **nearest neighbors** of the **query embedding** that represents the user's request.

Many developers want support for storing and indexing vector embeddings in their operational databases to more easily integrate these new capabilities into existing applications and development workflows. They also expect:

- transactional semantics and strong consistency between the original data, their vector representations and the vector indices
- SQL-based vector search that includes unified SQL access to both conventional structured data and vector data/indices, so that they can combine filters, joins and other SQL operations with vector search

PostgreSQL has long been one of the most popular operational databases, and the recent introduction of the pgvector extension for SQL-based vector capabilities has made it one of the most popular vector databases as well. The **pgvector** extension introduced a new vector data type, vector distance operations using SQL, and two types of indexes to PostgreSQL. Among the two, the most popular has been the **Hierarchical Navigable Small Worlds (HNSW) ANN index**, launched in August 2023.

Google Cloud supports pgvector in both Cloud SQL PostgreSQL and AlloyDB for PostgreSQL, and many of our customers have adopted the HNSW index to accelerate their vector search. However, the HNSW index doesn't always work for customers' real-world workloads, which may include large datasets or require faster index creation, faster vector search, faster real-time writes and/or updates, or a low memory footprint.

AlloyDB's launch of the pgvector-compatible ScaNN for AlloyDB index addresses these needs. In particular, when compared to the HNSW index in standard PostgreSQL, ScaNN for AlloyDB:

8×

offers up to 8× faster index creation

4×

offers up to 4× faster vector queries

3–4×

typically uses 3–4× less memory

10×

offers up to 10× higher write throughput

Source: Google Internal Data, April 2024.

By introducing ScaNN for AlloyDB alongside pgvector HNSW, AlloyDB provides customers the largest set of options in the PostgreSQL world, from which they can choose what fits their needs best.

This whitepaper explains how ScaNN for AlloyDB achieves these important performance and memory footprint benefits. The short answer: AlloyDB is leveraging vector search technology that has been developed over the last 12 years by Google Research, with ScaNN being the vector search technology behind Google Search, Youtube, and Ads. We will:

- **cover the basics of tree-quantization based indices like ScaNN**
- **demonstrate how ScaNN improves vector indexing and search in AlloyDB, especially compared to graph-based algorithms like HNSW, and**
- **explain key innovations in ScaNN that help it excel**

Part 1: The basics of nearest neighbor search

Part 2: The basics of ScaNN

Index construction

Querying

Putting it together in AlloyDB

Part 3: Performance of ScaNN

Part 4: Comparison with graph-based indices and HNSW

Comparison on ease of management and cost control

Memory footprint and indexing performance

Query performance

Part 5: Key innovations

CPU friendliness and SIMD

The power of geometry awareness

Anisotropic vector quantization (AVQ)

Spilling with Orthogonality Amplified Residuals (SOAR)

Conclusion

Part 1:
The basics of nearest neighbor search

Before we begin the discussion of ScaNN, let's review Approximate Nearest Neighbor (ANN) search and the role of vector indices. Finding the *k* nearest neighbors is called the *kNN* problem, which has an obvious brute force solution: compute the distance between each database embedding and the query embedding and return the top *k* database embeddings that led to the shortest distances. Depending on the application, typical distance metrics include euclidean, cosine, and inner product. But the brute force solution doesn't scale to millions or billions of vectors as brute forcing this search would be latency and cost prohibitive. This is where **Approximate Nearest Neighbor (ANN) vector indices** enter the picture. By indexing the database embeddings in ANN indices, smart ANN search algorithms *approximately* find the *k* nearest neighbors order(s) of magnitude faster.

Notice the use of approximation: the search algorithm may not return the exact *k* nearest neighbors. Occasionally, a few database embeddings that aren't truly among the *k* nearest neighbors will sneak into the top-k. This is called **recall loss**. While recall loss generally isn't ideal for the application, the majority of use-cases can tolerate some small recall loss — especially given the performance benefit it enables. For example, in a retail application offering product search, a small recall loss may occasionally cause a few less-relevant Stock Keeping Units (SKUs) to be returned, but this is acceptable since the vast majority of the returned items are highly relevant to the user request.

Furthermore, the embedding models themselves also create recall loss because even the best models can't perfectly capture the meaning of text and images. As we can see in the broader market, small recall losses haven't hindered the delivery of amazing applications in the Gen AI era but large recall losses can impact user experiences.

The important achievement of ANN indices, which has made them ubiquitous, is that they can achieve order(s) of magnitude speed improvement with minimal recall loss. For example, a typical target of 95% recall (equivalent to 5% recall loss) means that in, say, the top 20 results, on the average just one of the selected vectors won't be a true top 20 nearest neighbor.

While the ANN space has produced multiple innovations, it is far from being a fully solved problem. As the applications grow in usage and their data grow ever bigger, users demand higher performance, better index update throughput, low memory footprint, and fast index creation. In addition, database users expect vector indexing in their SQL environment so that they access vectors along structured data with SQL and have the usual guarantees of transactional consistency.

Part 2:
The basics of ScaNN

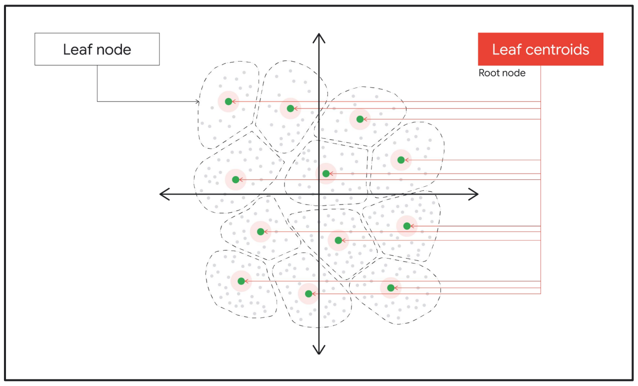
ScaNN (Scalable Nearest Neighbor) is the fundamental Google Research technology that AlloyDB uses for vector indexing. The release of ScaNN for AlloyDB leverages some of the ScaNN library techniques to accelerate vector index creation, to accelerate vector search and to deliver low memory footprint. We'll cover the basics here but, for those interested in additional details, there are many articles on ScaNN published in prestigious research publications, including ICML and NeurIPS papers. ScaNN is also available in the Google Research Github repository.

ScaNN indices belong to the family of **tree-quantization based ANN indices**. In a nutshell, tree-quantization techniques learn a search tree together with a quantization (or hashing) function. When querying, the search tree is used to prune the search space while quantization is used to compress the index size and, thereby, speed up the scoring of the similarity (i.e., distance) between the query vector and the database vectors. We refer interested readers to this journal paper for a comprehensive survey on the topic. In the interest of presentation simplicity, we describe first the two-level tree and set aside the important Asymmetric Hashing (AH), Anisotropic Vector Quantization (AVQ) and spilling (SOAR) optimizations for later in this whitepaper.

Index construction

A two-level tree is made up of a root and of leaf nodes that branch from that root. Each leaf node includes a set of vectors that are close together in *n*-dimensional space. Each leaf node is associated with a **centroid** vector. The root of the tree comprises a list of all of the **centroid** vectors and pointers to the respective leaf nodes.

During index construction, centroids are first **trained** using sample vectors and a modified k-means algorithm. The centroids are chosen to (approximately) minimize the distances of the vectors from their closest centroid, i.e., from the centroid of their leaf node. Each vector *v* is then placed on the leaf node whose centroid is closest to the vector *v*. In that sense, each leaf corresponds to the part of the space that is closer to its centroid than to any other centroid. The illustration on the next page shows a two-level index over two-dimensional vectors. Of course, in practice, the dimensions are far more than two.



A two-level index over two-dimension vectors

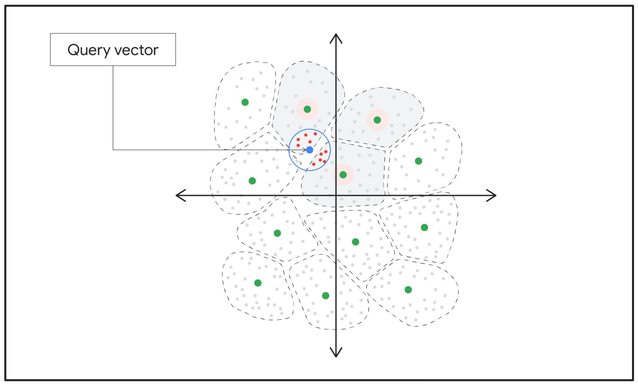
The two-level tree idea generalizes to multiple levels: in a three-level tree, each second level node (often called **branch node**) has its own centroid and contains all the leaf level centroids that are closest to it, along with pointers to the leaf nodes.

Querying

Consider an example query. This query finds the 20 tickets whose embedding vectors are closest to the provided parameter, which we call the **query embedding** (aka **query vector**).

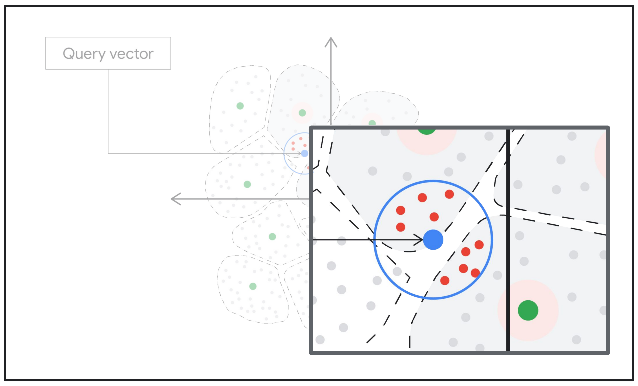
```
SELECT id, user_text, incident_time
FROM tickets
ORDER BY embedding <-> ?
LIMIT 10
```

At query time, under the hood, AlloyDB `ScaNN` will compute the distance of each centroid from the query vector — note the discussion still assumes a two-level tree. The leaves corresponding to the closest centroids will be searched and the others won't. The session parameter `scann.num_leaves_to_search` specifies how many leaf nodes will be searched. Let's assume for this example that `scann.num_leaves_to_search`= 3. In the next diagram, our query vector is highlighted in blue and the three leaves chosen to search are in grey. That's because their centroids were closest to the query vector.



The centroids of the grey highlighted leaf nodes are closest to the (blue) query vector

Next, AlloyDB will compute the distance between the query vector and each vector in the selected leaf nodes and maintain a top-10 list, implemented using a priority queue. The final result is the 10 vectors within the blue circle. As is the case with any ANN index, the result may entail some recall loss. For instance, when we zoom in on the running example (below) we see that one of the exact top 10 isn't included in the result because it belongs to a non-selected leaf node.



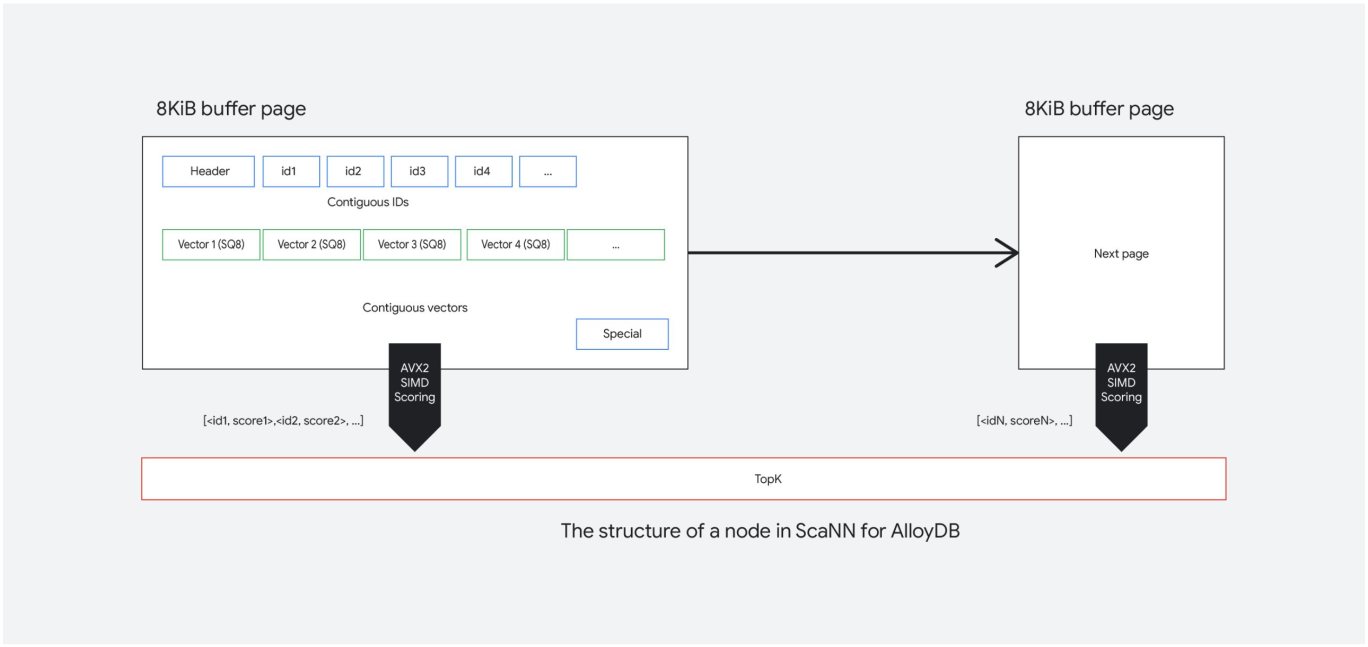
Below and to the left of the query vector there is a database vector that should be in the top-10 but its leaf wasn't selected

The obvious way to improve recall is to raise the `scann.num_leaves_to_search`(number of leaf nodes that are searched). However, this has the expected tradeoff effect on latency and queries per second (QPS). This is why benchmarks present recall-QPS curves, where the QPS falls as the recall goes up. ScaNN has an ace up its sleeve on raising the recall, without raising the `scann.num_leaves_to_search`.But before we get on to such advanced techniques, let's introduce how we incorporated the fundamental ScaNN algorithm into the ScaNN for AlloyDB index.

Putting it together in AlloyDB

ScaNN for AlloyDB brings the fundamental ScaNN index and algorithm into AlloyDB. This entails expanding it to work for large datasets, which exceed the memory size, by storing the tree in buffer pages like all other indexes. The updating of the index pages is transactionally consistent and thus the database user needn't worry about whether the results of vector search reflect the latest state of the database.

Presently, the float32 vectors of pgvector are quantized into SQ8 (scalar quantized int8) vectors, which use 8 bits per dimension. This reduces the index size and accelerates processing. In particular, the vectors are laid out in a contiguous fashion in storage to enable efficient distance computations. Each leaf node consists of a chain of buffer pages containing these contiguous SQ8 vectors. Root and branch nodes are laid out in the same way.



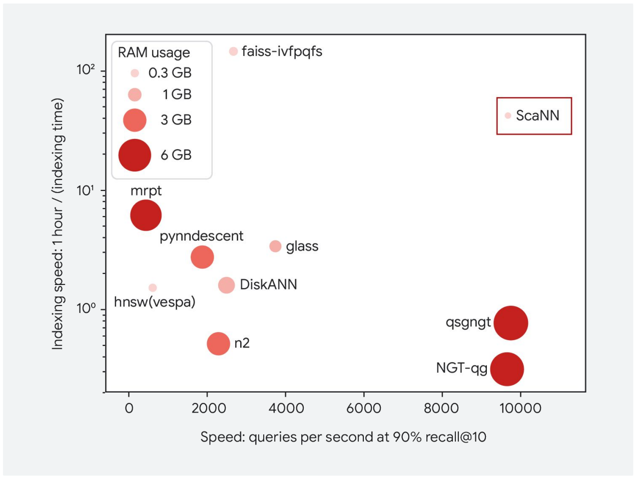
The structure of a node in ScaNN for AlloyDB

Part 3:
Performance of ScaNN

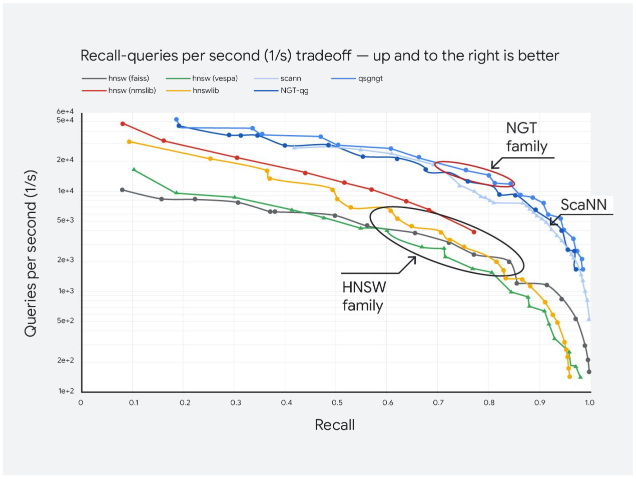
As we discussed above, ScaNN for AlloyDB outperforms the graph-based pgvector HNSW in a large number of cases: up to 8× faster index creation, typically 3–4× smaller memory footprint, write throughput up to 10× higher, and vector queries that are up to 4× faster — despite not yet having incorporated all of the techniques of the fundamental ScaNN algorithm.

But more generally, the fundamental ScaNN algorithm typically outperforms graph-based algorithms in benchmarks. Let's take a look.

The ScaNN library released by Google Research runs on x86 CPUs with AVX2 support and the index is in main memory. (Note, ScaNN for AlloyDB doesn't require that the index fits in main memory.) In this main memory setting, ScaNN outperforms (in QPS, queries per second) all HNSW main memory implementations while using 2–5× less memory. The only family that provides competitive search performance for relatively small datasets, the NGT family, has a 14× larger memory footprint and correspondingly larger index building times.



A more detailed performance picture below shows the results of the standard ANN-Benchmarks, the most popular benchmark for ANN implementations. These benchmarks plot the tradeoffs between recall and latency. Up and to the right is better: it means an implementation has higher recall at a similar QPS and higher QPS at a similar recall percentage.



Comparison of queries per second for multiple recalls. Up and to the right is better.

Where do these performance gains come from? We dive deep into this question, starting with a comparison to graph-based indices.

Source: Google Internal Data, April 2024

Part 4: Comparison with graph-based indices and HNSW

Let us start with a quick primer of graph-based indexing. The pgvector HNSW index belongs to the family of graph-based indices. While graph-based indices come in multiple variations, they all share a common theme: the index is a graph whose nodes are the database vectors and whose edges connect neighboring database vectors, i.e., neighboring nodes. At search time, navigations start from entry points in the graph and move in the directions that minimize the distance to the query vector. HNSW maintains a priority queue, of size `ef_search`, that contains the nodes found to be nearest to the query vector thus far. As it traverses the graph, the algorithm evaluates the neighbors of the current node to see if any of them are closer than the nodes (i.e., database vectors) on the list so far. If so, these neighbors are added to the queue and it continues traversing the graph. The algorithm terminates when none of the neighbors of the nodes in the priority queue are closer to the query vector than the nodes in the priority queue themselves. Therefore, the `ef_search` plays a crucial role in the recall/QPS tradeoff: higher `ef_search` increases the number of navigations that are pursued, thus raising both the recall and the latency.

This table presents a summary of the ScaNN and HNSW approaches, while also making broader comments about other members of the graph-based family. We detail these points below.

Method	**ScaNN for AlloyDB**	**pgvector HNSW**
Family	Tree-quantization based indexing. Also includes Faiss, SPTAG.	Graph-based indexing. Also includes HNSW, NGT, DiskANN (Vamana).
Index size	Tree overheads are typically smaller. Quantization further reduces size.	Graphs have more edge connections therefore bigger index overhead.
Index time	Tree training and indexing are faster. The operation is of O(#vectors * #centroids).	Graph construction fundamentally requires many vector–vector comparisons.
Memory access	Vectors in tree leaves are stored contiguously. Memory access is more continuous and friendly to SIMD acceleration.	Graph walk beam search requires random memory accesses. NGT partially solves this by duplicating data in nodes, but leads to bloated indices.
Latency	Constant search speed across queries. Typically configured to search a fixed number of leaves.	Varies more per query. Easy queries terminate early as nothing is left in the working set of beam search. But tail latency for harder queries could be higher.

Comparison on ease of management and cost control

From an ease-of-use and management perspective HNSW is harder to cost control. At a high level, one might assert that **"`ef_search` is to graph-based search what `scann.num_leaves_to_search` is to tree-based search"** because they both control recall. However, this isn't a fair comparison because `scann.num_leaves_to_search`comes with a performance guarantee. It produces predictably low latency (and thus QPS) because it specifies how many nodes are visited and nodes are of more-or-less similar size. In contrast, the HNSW convergence time is variable: it may be lucky and quickly reach the result nodes, or it may go into long sequences of replacing the candidates in the priority queue with just-a-little-bit-better candidates and thus reaching the convergence point much slower.

Memory footprint and indexing performance

ScaNN has much lower memory footprint than HNSW for a simple fundamental reason: in addition to the vectors themselves, ScaNN only stores the centroids — which are far fewer than the vectors themselves. Since >100 vectors per leaf node is a reasonable aim, it follows that the centroid overhead is around 1% of total vector size. In contrast, HNSW has to represent multiple edges per node. For example, the HNSW index that is tuned for the Glove-100 ANN-benchmark has 20 edges per node.

The difference between the memory footprints of the indices has to be carefully considered when the customer chooses between ScaNN for AlloyDB and HNSW: it may be the case that the ScaNN index is small enough to be cached in memory, while only a small part of the HNSW index will be cached. Then, the HNSW performance will become IO-bound while ScaNN will be CPU-bound. Moreover, in cases where both HNSW and ScaNN are IO-bound, the distinction between random IO access vs. sequential IO access is critical. Furthermore, real workloads are mixed — they will typically not just be pure vector search. The other parts of the workload also benefit from more memory.

The smaller size of the index also pays off on indexing time. Furthermore, the index construction algorithm of ScaNN is running a modified k-means algorithm, thus having O(#centroids * #vectors) algorithmic complexity rather than the graph algorithms, which compare many pairs of vectors. Indeed, generally (and unsurprisingly) the tree-quantization family at large does better on index size and indexing time than the graph family.

Query performance

Even the first version of ScaNN for AlloyDB, with only a subset of ScaNN techniques, can beat HNSW by up to 4× in query performance in benchmarks. Similarly, the ScaNN library outperforms the graph-based family on main memory implementations, with notable partial exceptions of

the NGT sub-family that ties/beats ScaNN, but with 14× bigger memory footprint and

performance at extremely high recalls (98%+) for moderately-sized data where graph-based algorithms occasionally outperform ScaNN, but the latter regains its advantage when it is instructed to employ spilling. With spilling, each vector is placed in multiple (usually two) positions in the index

To graph aficionados these query performance results may appear counterintuitive. After all, the larger size of the graph-based indices (which is due to the many edges) should confer an advantage to graph-based algorithms: there are multiple paths leading to a result node. If a path from an entry point towards a result node gets stuck in a local minimum of distance from the query vector, there is still a second path, a third path, and many more paths that may fare better. In contrast, tree-based algorithms have a single path from the root to each result node and ScaNN with spilling is usually configured to have two paths, essentially placing each vector in two leaf nodes. How can they do even better? At a high level, the answer is twofold:

Geometry awareness: while graph-based algorithms are only aware of distances, ScaNN is aware of the geometry behind these distances and exploits this knowledge to (a) produce better clustering and (b) to strategically position the (usually) two copies of each vector so that if one copy fails to be retrieved, the other copy will have an extremely high chance of being retrieved.

***CP*U-friendliness: ScaNN is friendly to modern CPU architectures and, in particular, to their parallelism using Single Instruction Multiple Data (SIMD). This enables "brute forcing" the evaluation of many more candidates.**

Part 5: Key Innovations

Next, we elaborate on these geometry aware and CPU-friendly innovations and introduce the SOAR, AVQ and AH techniques employed by ScaNN.

CPU friendliness and SIMD

The vectors in tree leaves are stored in contiguous memory space and on-disk, which is friendly to the large caches and SIMD abilities of modern CPUs. In contrast, HNSW has to, in each step, access neighboring vectors that are randomly placed in memory. Interestingly, the NGT sub-family of graph-based indexing avoids HNSW's random access problem by having copies of neighboring vectors on each node/vector. Thus, NGT achieves much higher query performance — winning or losing against ScaNN by a small margin for small datasets — but at the price of 14× larger memory footprint than ScaNN, which precludes the use of NGT for non-small datasets. Moreover, we aren't aware of a PostgreSQL implementation of NGT.

The utilization of SIMD is further optimized by quantizations, which are **compact approximate representations of the vectors**. Every quantization speeds up the computation of the distance between a candidate vector (i.e., a vector in one of the selected leaf nodes) and the query vector, while causing minimal recall loss. The present release of ScaNN for AlloyDB uses the SQ8 quantization, which uses 8 bits per dimension. It turns out that the recall loss caused by the fewer bits is usually around 1% for modern use cases.

The Asymmetric Hashing (AH) feature of ScaNN is a far more sophisticated quantization, though it isn't yet part of ScaNN for AlloyDB. Asymmetric Hashing entails splitting each vector into segments[1] and each segment is approximated by the closest vector in a so-called **code book**. By doing so, ScaNN no longer engages in the expensive multiplications between query vector *q* and each candidate vector *v*. Rather, it multiplies segments of the query vector with the segments in the code book, thus producing a **query code book**, and then it approximates the *q* * *v* product by looking up and summing the relevant pieces from the query code book. It turns out that since 2013 CPUs can do this lookup in parallel. It is a testimony to the AH efficiency that the ScaNN deployments systematically find that their bottleneck is the main memory bandwidth and not the (SIMD optimized) CPU operations!

The power of geometry awareness

ScaNN has geometric awareness of the nature of the distances. This awareness is evident in optimizations that guide the placement of edges in the tree. Indeed, it is interesting to contrast these geometry aware placements with the purely distance-minded placement of edges by graph-based algorithms. We will first describe the Anisotropic Vector Quantization (AVQ) and its initially surprising, yet intuitive once you account for geometry, answer to the question "**to which leaf/centroid *c* I should connect a vector *x***". Then we will discuss spilling and its geometry aware way of essentially creating two paths to each vector, by smartly placing a second copy of each vector on another leaf.

[1]It also entails linearly transforming the vector. See here.

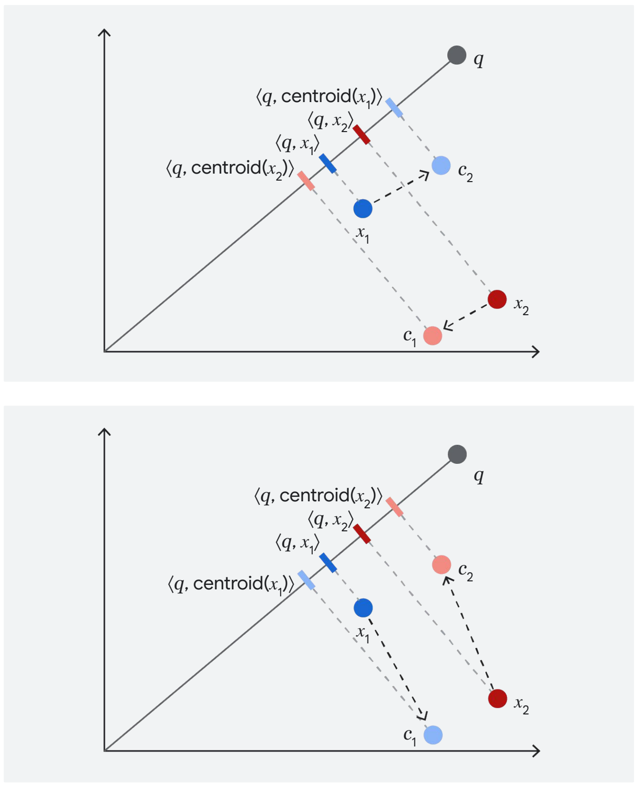
Anisotropic vector quantization (AVQ)

The classic approach of tree-quantization indices was to place each vector on the leaf node of the nearest centroid. This is based on the premise that the distance between the query vector and the database vector is usually well approximated by the distance between the query vector and its nearest centroid. However, the existence of recall loss tells us that this approximation isn't always perfect; sometimes a database vector that is itself very close to the query vector (and thus should be in the result) is missed because its centroid isn't so close to the query vector and other centroids/leaves are chosen.

Anisotropic Vector Quantization (AVQ) produces a more efficient assignment of vectors to leaves/centroids by having a key intuition that the classic approach ignores: most vector searches are looking for a tiny percentage of very relevant vectors among millions or billions in the database. The query vector, the chosen centroids, and the exact result vectors are thus close. Therefore, we care that a centroid best approximates the distance between its leaf vectors and query vectors in the case where the query vectors are very close. Note that this is different from the classic approach, which essentially optimizes the average approximation of distance over all possible query vectors as if all query vectors (both the near ones and the far ones) matter the same. They don't!

To see the difference between the classic approach and the AVQ approach, consider the example in the diagram on the right. Suppose we have an index to assist in vector search using inner product. We have the two vectors x_1 and x_2 of the image on the right and we need to assign each one either to a leaf with centroid c_1 or to a leaf with centroid c_2. The classic answer to this decision problem is to connect x_1 to centroid(x_1) = c_2 and connect x_2 to centroid(x_2) = c_1 because x_1 is closer to c_2 and x_2 is closer to c_1. But this is suboptimal! The optimal connections are x_1 to c_1 and x_2 to c_2. To realize why, think of the query vector q, which is close to these vectors and centroids.

The goal is to make the inner product <*q, centroid*(x_i)> as similar to the inner product <q, x_i> as possible so that the best centroid/leaves are chosen. This can be visualized as making the magnitude of the projection of centroid(x_i) onto q as similar as possible to the projection of x_i onto q. This is illustrated in the figure below and leads to the opposite result: centroid(x_1) = c_1 and centroid(x_2) = c_2.



q
$\langle q, \text{centroid}(x_1)\rangle$
$\langle q, x_2\rangle$
$\langle q, x_1\rangle$
$\langle q, \text{centroid}(x_2)\rangle$
c_2
x_1
x_2
c_1
q
$\langle q, \text{centroid}(x_2)\rangle$
$\langle q, x_2\rangle$
$\langle q, x_1\rangle$
$\langle q, \text{centroid}(x_1)\rangle$
c_2
x_1
x_2
c_1

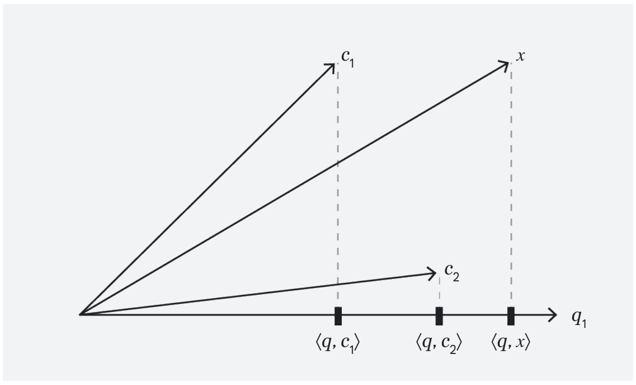
Assigning vectors to the nearest centroid isn't always right!

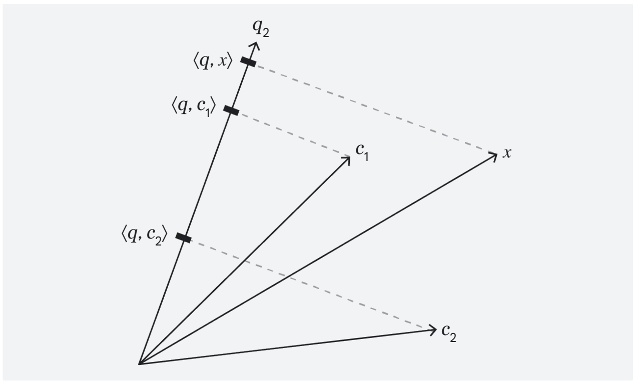
We see that direction matters as well as magnitude — even though c1 is farther from x_1 than c_2, c_1 is offset from x_1 in a direction almost entirely orthogonal to x_1, while c_2's offset is parallel (for x_2, the same situation applies but flipped). It turns out that error in the parallel direction is much more harmful in the inner product vector search because it disproportionately impacts high inner products, which by definition are the ones that we want to estimate as accurately as possible. Thus, ScaNN indices more significantly penalize the parallel distance from the centroid.

Spilling with Orthogonality Amplified Residuals (SOAR)

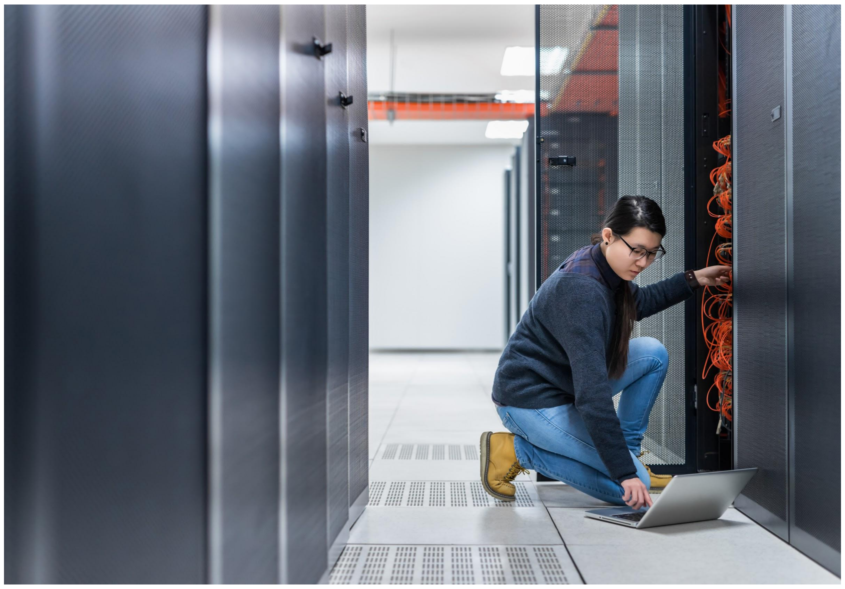
While AVQ improves clustering, there will be some queries that have nearest neighbors that are very difficult to find. ScaNN for AlloyDB uses the SOAR technique to circumvent this problem via redundancy: it places each database vector *x* into two leaf nodes so that when a query vector *q* (that is a near neighbor of *x*) fails to find the first leaf, the probability that it will find the second leaf becomes much higher. Note, this isn't a naive application of redundancy — as it would be the case if ScaNN simply chose the two nearest centroids to the database vector *x* and placed *x* in the respective leaf nodes. Rather, after ScaNN inserts the database vector in a first leaf, the SOAR technique chooses the second leaf in a geometry aware way that maximizes the chances that the second leaf will be found by queries that missed the first leaf.

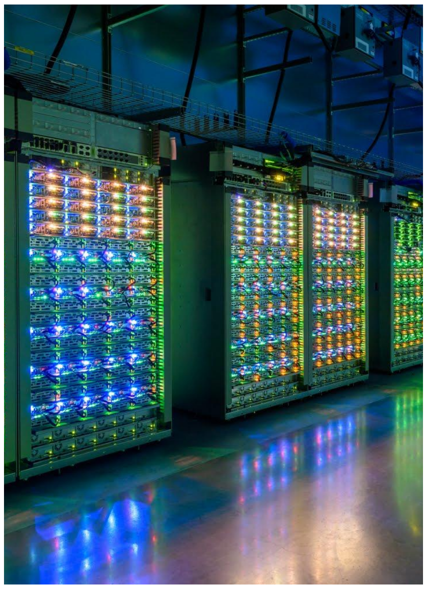
To understand the choice of the two placements, it is worth understanding why there will always be some query vectors *q* that will miss a database vector *x* even if *x* is very close to *q*. Consider centroids c_1 and c_2, the database vector *x*, two queries q_1 and q_2 (as in the images on the right) and inner product distance. For q_1, c_2 provides a better inner product estimate (i.e., $<q_1,c_2>$ is a better estimate of $<q_1,x>$ than $<q_1,c_1>$). But for q_2 it is the other way around.





Essentially, the approximation of each database vector by its assigned centroid is at its worst when the vector-centroid residual is parallel to the query. Therefore, SOAR chooses leaf nodes to assign the database vector in a way that maximizes the probability that when the vector-first centroid residual is parallel for a query, the vector-second centroid residual will be more orthogonal to this query and, thereby, provide a better approximation. (It is good news that in high-dimensional spaces, unlike our 2D examples, there are plenty of ways to be orthogonal to a direction!)





Conclusion

AlloyDB now gives you the richest set of native vector search options in SQL databases, by offering both the graph-based pgvector HNSW and the pgvector-compatible ScaNN for AlloyDB based on tree quantization. This is just the beginning of a journey which brings these and other vector innovations to our AlloyDB users.