Vector databases (Part 3): Not all indexes are created equal

July 24, 2023 18-minute read
databases
vector-db

Organizing vector indexes

组织向量索引

This is my third post in a series on vector databases. Part 1 compared the offerings of various DB vendors and how they are different at a high level, while Part 2 focused on the basics of what vector DBs are and what they do. You may have already come across the excellent post "*Not all vector databases are made equal*"[1] by Dmitry Kan, which covered the differences between various vector DBs in the market back in 2021. The landscape has been continuously evolving since then, and because each DB is different from others in its internals, I thought it made sense to do a deeper dive into indexes, the backbone of vector search.

这是我关于矢量数据库系列文章的第三篇。第 1 部分比较了不同数据库供应商的产品以及它们在高级别上的不同之处，而第 2 部分则重点介绍了矢量数据库是什么以及它们的作用的基础知识。您可能已经读过 Dmitry Kan 的优秀文章“并非所有矢量数据库都是一样的”[1]，该文章涵盖了 2021 年市场上各种矢量数据库之间的差异。从那时起，环境一直在不断发展，并且由于每个数据库的内部都与其他数据库不同，我认为更深入地研究索引（矢量搜索的支柱）是有意义的。

Assuming that it's amply clear to you what a vector database *is*, it's worth taking a step back to wonder, how does it all scale so wonderfully to be able to search millions, billions, or even trillions of vectors[2]? The primary aim of a vector database is to provide a fast and efficient means to store and *semantically* query data, in a way that the `Vector` data type is a first-class citizen. The similarity between two vectors is gauged by distance metrics like cosine distance or the dot product. When working with vector databases, it's important to distinguish between the *search algorithm*, and the underlying *index* on which the Approximate Nearest Neighbour (ANN) search algorithm operates.

假设您非常清楚什么是矢量数据库，那么值得退后一步来思考一下，它是如何如此出色地扩展以能够搜索数百万、数十亿甚至数万亿个矢量[2]的？矢量数据库的主要目的是提供一种快速有效的方法来存储和语义查询数据，使 `Vector` 数据类型是一等公民。两个向量之间的相似性由距离度量（如余弦距离或点积）来衡量。使用矢量数据库时，区分搜索算法和近似最近邻 （ANN） 搜索算法运行的基础索引非常重要。

As in most situations, choosing a vector index involves a tradeoff between accuracy (precision/recall) and speed/throughput. Having scoured the literature, I find that vector indexing methods can be organized within two levels: by their data structures, and by their level of compression. These classifications are by no means exhaustive, and many sources disagree on the right way to organize the various indexes, so, this is my best attempt at making sense of it all. Here goes! 😅

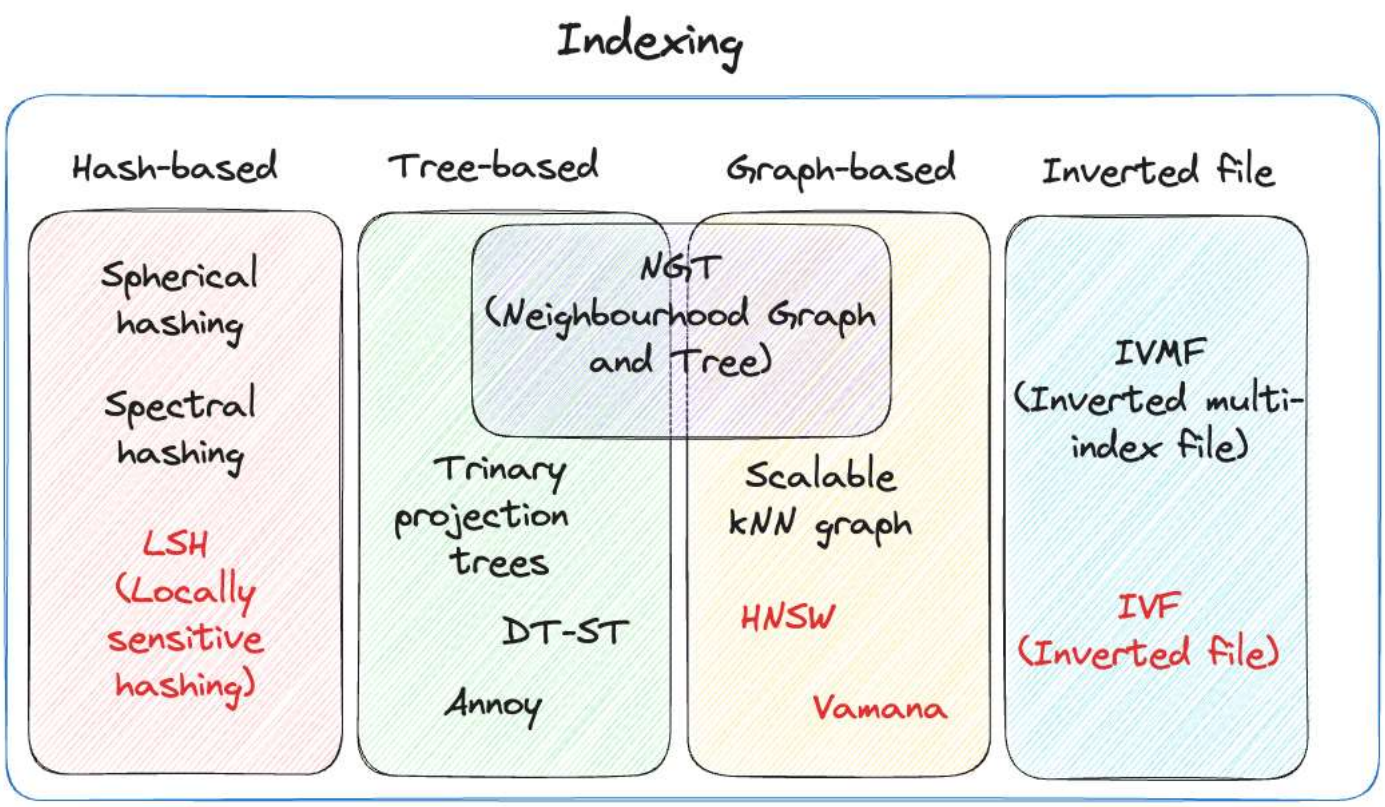
与大多数情况一样，选择矢量索引需要在准确性（精度/召回率）和速度/吞吐量之间进行权衡。在搜索了文献之后，我发现矢量索引方法可以分为两个级别：按数据结构和压缩级别。这些分类绝不是详尽无遗的，许多来源对组织各种索引的正确方法存在分歧，因此，这是我理解这一切的最佳尝试。来了！ 😅

Level 1: Data structures

级别 1：数据结构

It helps to start by organizing indexes based on the data structures that are used to construct them. This is best explored visually.

首先，根据用于构造索引的数据结构组织索引会有所帮助。最好通过视觉来探索这一点。



Breaking down vector indexes by their underlying data structures

按底层数据结构分解向量索引

Hash-based index 基于哈希的索引

Hash-based indexes like LSH (Locally Sensitive Hashing) transform higher dimensional data into lower-dimensional hash codes that aim to keep the original similarity as much as possible. During indexing, the dataset is hashed multiple times to ensure that similar points are more likely to collide (which is the opposite of conventional hashing techniques, where the goal is to minimize collisions). During querying, the query point is also hashed using the same hash functions as used during indexing, and because the similar points are assigned to the same hash bucket, retrieval is very fast. The main advantage of hash-based indexes is that they are very fast while scaling to huge amounts of data, but the downside is that they are not very accurate.

LSH（本地敏感哈希）等基于哈希的索引将高维数据转换为低维哈希代码，旨在尽可能保持原始相似性。在索引编制期间，数据集被多次散列，以确保相似的点更有可能发生冲突（这与传统的散列技术相反，后者的目标是最大限度地减少冲突）。在查询期间，查询点也使用与索引期间相同的哈希函数进行哈希处理，并且由于相似的点分配给同一哈希桶，因此检索速度非常快。基于哈希的索引的主要优点是它们在扩展到大量数据时非常快，但缺点是它们不是很准确。

Tree-based index 基于树的索引

Tree-based index structures allow for rapid searches in high-dimensional spaces, via binary search trees. The tree is constructed in a way that similar data points are more likely to end up in the same subtree, making it much faster to discover approximate nearest neighbours. **Annoy** (Approximate Nearest Neighbours Oh Yeah) was such a method that uses a forest of binary search trees, developed at Spotify. The downside of tree-based indexes is that they perform reasonably well only for low-dimensional data, and are not very accurate for high-dimensional data because they cannot adequately capture the complexity of the data.

基于树的索引结构允许通过二叉搜索树在高维空间中快速搜索。树的构造方式是，相似的数据点更有可能出现在同一个子树中，从而更快地发现近似的最近邻。烦恼（近似最近的邻居哦，是的）是一种使用二叉搜索树森林的方法，由Spotify开发。基于树的索引的缺点是，它们仅对低维数据表现得相当好，而对于高维数据则不是很准确，因为它们无法充分捕获数据的复杂性。

Graph-based index 基于图形的索引

Graph-based indexes are based on the idea that data points in vector space form a graph, where the nodes represent the data values, and edges connecting the nodes represent the similarity between the data points. The graph is constructed in a way that similar data points are more likely to be connected by edges, and the ANN search algorithm is designed to traverse the graph in an efficient manner. The main advantage of graph-based indexes, is that they are able to find approximate nearest neighbours in high-dimensional data, while also being memory efficient, increasing performance. HNSW and Vamana, explained below, are examples of graph-based indexes.

基于图的索引基于向量空间中的数据点形成图的思想，其中节点表示数据值，连接节点的边表示数据点之间的相似性。该图的构造方式是相似的数据点更有可能通过边缘连接，并且ANN搜索算法旨在以有效的方式遍历图。基于图形的索引的主要优点是，它们能够在高维数据中找到近似的最近邻，同时还具有内存效率，从而提高了性能。下面解释的HNSW和Vamana是基于图形的索引的示例。

An extension of graph-based indexes to include concepts from tree-based indexes is NGT[3] (Neighbourhood Graphs and Trees). Developed by Yahoo! Japan Corporation, it performs two constructions during indexing: one that transforms a dense kNN graph into a bidirectional graph, and another that incrementally constructs a navigable small world (NSW) graph. Where it differs from pure graph-based indexes is in its use of range search via a tree-like structure (Vantage-point, or 'VP' trees), a variant of

greedy search during graph construction. Because both constructions result in nodes that have a high outward degree, to avoid combinatorial explosion, the seed vertex from which the search originates uses the range search to make the traversal more efficient. This makes NGT a hybrid graph and tree-based index.

基于图的索引的扩展包括来自基于树的索引的概念是NGT [3] （邻域图和树）。它由雅虎日本公司开发，在索引过程中执行两种构造：一种是将密集的 kNN 图转换为双向图，另一种是增量构造可导航的小世界 （NSW） 图。它与纯基于图的索引的不同之处在于它通过树状结构（Vantage-point或“VP”树）使用范围搜索，这是图构建过程中贪婪搜索的一种变体。由于这两种构造都会导致节点具有较高的向外度，因此为了避免组合爆炸，搜索源自的种子顶点使用范围搜索来提高遍历效率。这使得 NGT 成为基于树的混合图和索引。

Inverted file index 倒排文件索引

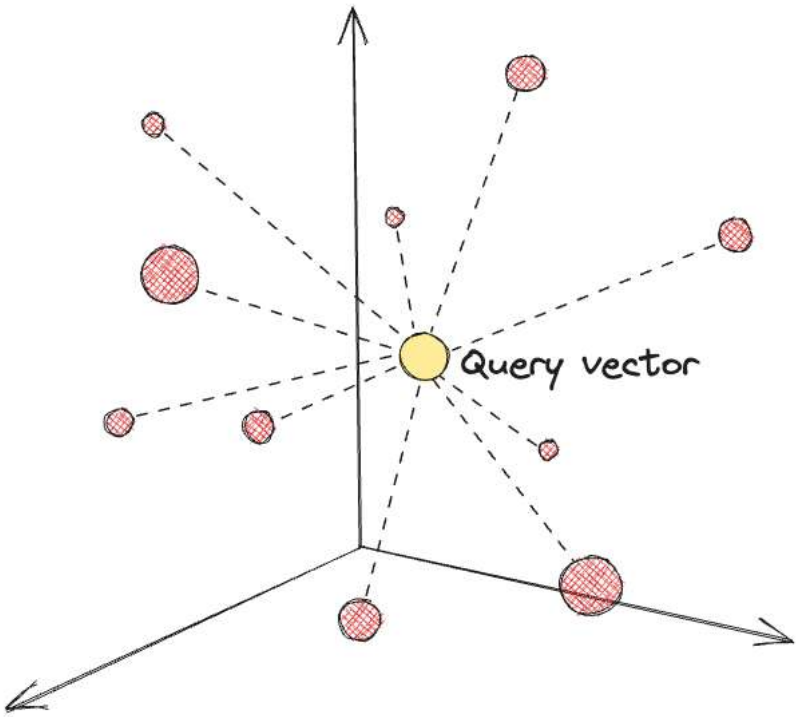
Inverted file index (IVF) divides the vector space into a number of tessellated cells, called Voronoi diagrams – these reduce the search space in the same way that clustering does. In order to find the nearest neighbours, the ANN algorithm must simply locate the centroid of the nearest Voronoi cell, and then search only within that cell. The benefit of IVF is that it helps design ANN algorithms that rapidly narrow down on the similarity region of interest, but the disadvantage in its raw form is that the quantization step involved in tessellating the vector space can be slow for very large amounts of data. As a result, IVF is commonly combined with quantization methods like product quantization (PQ) to improve performance, described below.

倒置文件索引（IVF）将向量空间划分为许多细分单元格，称为Voronoi图-这些单元以减少搜索空间的方式与聚类相同。为了找到最近的邻居，ANN算法必须简单地定位最近的Voronoi单元的质心，然后仅在该单元内搜索。IVF的好处是它有助于设计ANN算法，快速缩小感兴趣的相似性区域，但其原始形式的缺点是，对于非常大量的数据，细分向量空间所涉及的量化步骤可能很慢。因此，IVF通常与产品量化（PQ）等量化方法结合使用，以提高性能，如下所述。

Level 2: Compression 级别 2：压缩

The second level on which indexes can be organized is their compression level: a “flat” or brute force index is one that stores vectors in their unmodified form. When a query vector is received, it is exhaustively compared against each and every vector in the database, as shown in the simplified example below in 3-D space. In essence, using such an index would be like doing a kNN search, where the returned results are exact matches with the k nearest neighbouring vectors. As you can imagine, the time required to return results would increase linearly with the size of the data, making it impractical when applied on a dataset with more than a few hundred thousand vectors.

可以组织索引的第二个级别是它们的压缩级别： “平面”或暴力索引是以未修改的形式存储向量的索引。收到查询向量时，会将其与数据库中的每个向量进行详尽的比较，如下面的 3-D 空间中的简化示例所示。从本质上讲，使用这样的索引就像进行 kNN 搜索，其中返回的结果与 k 最近的相邻向量完全匹配。可以想象，返回结果所需的时间会随着数据大小而线性增加，因此在应用于具有数十万个向量的数据集时不切实际。

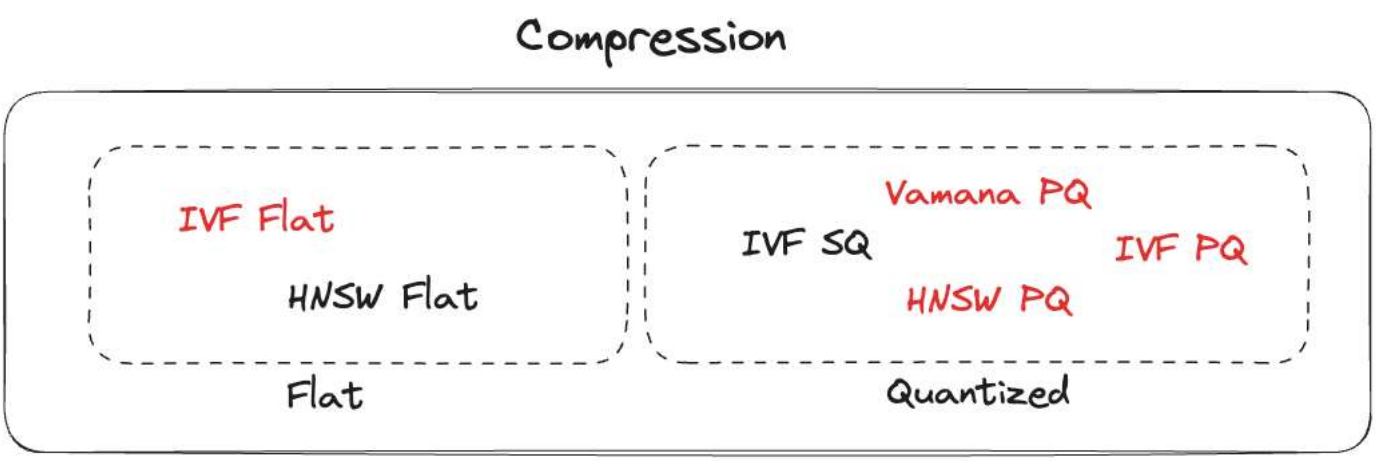


Flat index for kNN (exhaustive) search: Image inspired by Pinecone blog

kNN（详尽）搜索的平面索引：图片灵感来自松果博客

The solution to improve search efficiency, at the cost of some accuracy in the retrieval, is compression. This process is called *quantization*, where the underlying vectors in the index are broken into chunks made up of fewer bytes (typically via converting floats to integers) to reduce memory consumption and computational cost during search.

以牺牲检索的准确性为代价来提高搜索效率的解决方案是压缩。此过程称为量化，其中索引中的基础向量被分解为由较少字节组成的块（通常通过将浮点数转换为整数），以减少搜索期间的内存消耗和计算成本。



Flat indexes 平面索引

When using ANN (non-exhaustive) search, an existing index like IVF or HNSW is termed "flat" when it directly calculates the distance between the query vector and the database vectors in their raw form. To differentiate this from the quantized variants. When used this way, they are called IVF-Flat, HNSW-Flat, and so on.

当使用ANN（非穷举）搜索时，像IVF或HNSW这样的现有索引在直接计算查询向量和原始形式的数据库向量之间的距离时被称为“平面”。将其与量化变体区分开来。当以这种方式使用时，它们被称为IVF-Flat，HNSW-Flat等。

Quantized indexes 量化索引

A quantized index is one that combines an existing index (IVF, HNSW, Vamana) with compression methods like quantization to reduce the memory footprint and to speed up search. The quantization is typically one of two types[4]: Scalar Quantization (SQ), or Product Quantization (PQ). SQ converts the floating-point numbers in a vector to integers (which are much smaller in size in bytes) by symmetrically dividing the vector into bins that account for the minimum and maximum value in each dimension.

量化索引是将现有索引（IVF，HNSW，Vamana）与量化等压缩方法相结合的索引，以减少内存占用并加快搜索速度。量化通常是两种类型 [4] 之一：标量量化（SQ）或乘积量化（PQ）。SQ 通过将向量对称划分为占每个维度中的最小值和最大值的箱，将向量中的浮点数转换为整数（以字节为单位）小得多）。

PQ is a more sophisticated method that considers the distribution of values along each vector dimension, performing *both* compression and data reduction[4]: The idea behind PQ is to decompose a larger dimensional vector space into a cartesian product of smaller dimensional subspaces by quantizing each subspace into its own clusters – vectors are represented by short codes, such that the distances between them can be efficiently estimated from their codes, termed *reproduction values*. An asymmetric binning procedure is used (unlike SQ), which increases precision[5], as it incorporates the distribution of vectors within each subspace as part of the approximate distance estimation. However, there is a trade-off as it does reduce recall quite significantly[6].

PQ [4] 背后的思想是通过将每个子空间量化为自己的簇，将较大维向量空间分解为较小维子空间的笛卡尔乘积 - 向量由短代码表示， 这样就可以从它们的代码（称为复制值）有效地估计它们之间的距离。使用非对称分箱过程（与 SQ 不同），它提高了精度 [5]，因为它将每个子空间内的向量分布作为近似距离估计的一部分。但是，有一个权衡，因为它确实大大减少了召回 [6] 率。

Popular indexes 热门指数

Among all the indexing methods listed so far, most purpose-built vector databases implement only a select few of them. This is a very rapidly evolving space 🔥, so a lot of information here may be out of date when you're reading this. It's highly recommended you check out the latest documentation of the databases you're interested in, to see what indexes they support. In this section, I'll focus on a few popular and upcoming indexes that multiple vendors are focusing on.

在到目前为止列出的所有索引方法中，大多数专用矢量数据库只实现了其中的少数几种。这是一个发展非常迅速的空间🔥，所以当你阅读这篇文章时，这里的很多信息可能已经过时了。强烈建议您查看您感兴趣的数据库的最新文档，以了解它们支持哪些索引。在本节中，我将重点介绍多个供应商关注的一些流行和即将推出的索引。

IVF-PQ 试管婴儿-PQ

IVF-PQ is a composite index available in databases like Milvus and LanceDB. The IVF part of the index is used to narrow down the search space, and the PQ part is used to speed up the distance calculation between the query vector and the database vectors, and to reduce the memory requirements by quantizing the vectors. The great part about combining the two is that the speed is massively improved due to the PQ component, and IVF component helps improve the recall (that is normally compromised) by the PQ component alone.

IVF-PQ是Milvus和LanceDB等数据库中可用的复合索引。索引的IVF部分用于缩小搜索空间，PQ部分用于加快查询向量与数据库向量之间的距离计算，并通过量化向量来降低内存需求。将两者结合起来的重要部分是，由于PQ组件，速度大大提高，并且IVF组件有助于改善仅由PQ组件引起的召回（通常受到损害）。

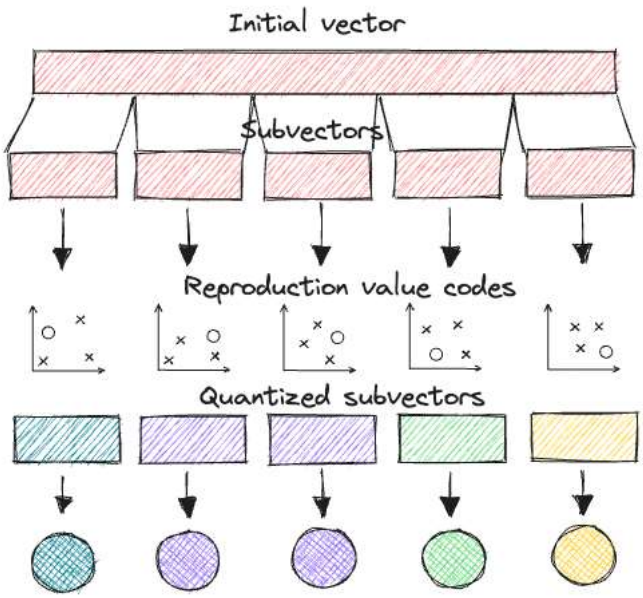
The PQ component can be broken down as per the diagram below. Each vector representing a data point consists of a fixed number of dimensions d (of the order of hundreds or thousands, depending on the embedding model used upstream). Because storing these many 32 or 64-bit floating point numbers can be quite expensive on a large dataset, product quantization approaches this problem in two stages: the first stage is a coarse quantization stage where the vector is divided into m subvectors, each

of dimension d/m, and each subvector is assigned a quantized value (termed “reproduction value”) that maps the original vectors to the centroid of the points in that subspace[7].

PQ 组件可以按照下图进行细分。表示数据点的每个向量由固定数量的维度 d 组成（大约数百或数千，具体取决于上游使用的嵌入模型）。由于在大型数据集上存储这些 32 位或 64 位浮点数可能非常昂贵，因此乘积量化分两个阶段解决此问题：第一阶段是粗量化阶段，其中向量被划分为 m 子向量，每个子向量都是维度 d/m 的，并且每个子向量被分配一个量化值（称为“再现值”），该值将原始向量映射到该子空间 [7] 中点的质心。

The second stage is similar to k-means clustering, where a “codebook” of values is learned by minimizing the distance between the original vector and the quantized vector centroids. By mapping a large, high-dimensional vector into smaller, lower-dimensional subvectors, it is only the codebook of quantized values that is stored, making the memory footprint far smaller.

第二阶段类似于 k 均值聚类，其中通过最小化原始向量和量化向量质心之间的距离来学习值的“密码本”。通过将大的、高维的向量映射到更小的、低维的子向量，它只存储量化值的密码本，使内存占用量小得多。

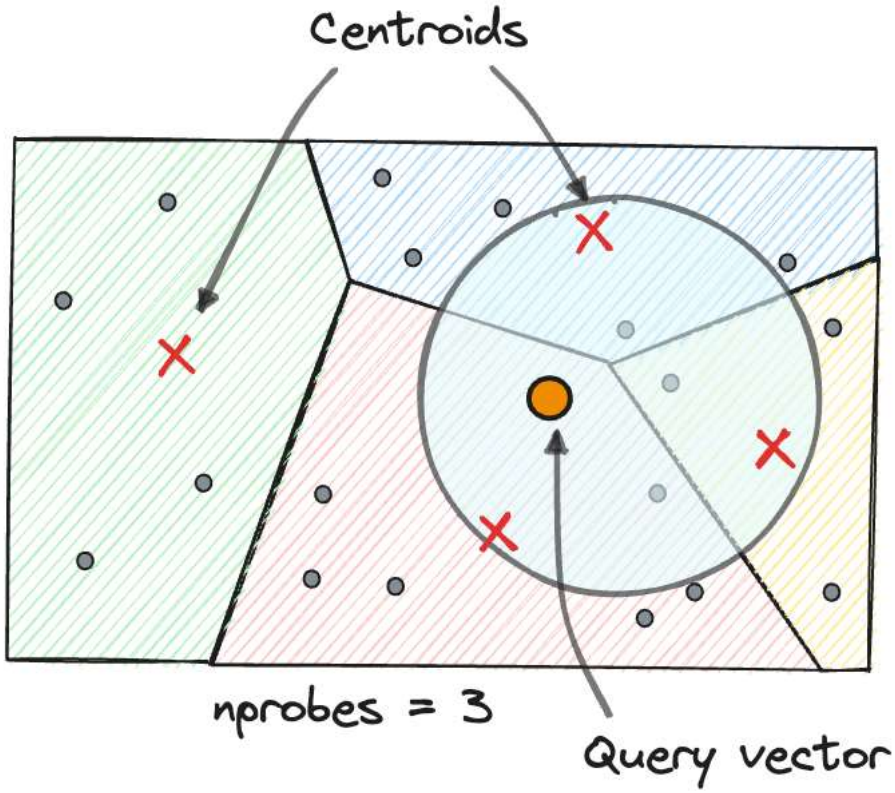


Product quantization: Image inspired by Pinecone blog

产品量化：图片灵感来自松果博客

IVF is then applied to the PQ vector space – for each point in the space there is a corresponding region, called a Voronoi cell, consisting of all points in the space closer to the source point (seed) than to any other. These seed points are used to create an inverted index that correlates each centroid with a list of vectors in the space.

然后将IVF应用于PQ向量空间 - 对于空间中的每个点，都有一个相应的区域，称为Voronoi单元，由空间中靠近源点（种子）而不是任何其他点的所有点组成。这些种子点用于创建倒排索引，该索引将每个质心与空间中的向量列表相关联。



Depending on where the query vector lands, it may be close to the border of multiple Voronoi cells, making it ambiguous which cells to return nearest neighbours from, leading to the *edge problem*. As a result, IVF-PQ indexes involve setting an additional parameter, `n_probes`, that tells the search algorithm to expand outward to the number of cells specified by the `n_probes` parameter.

根据查询向量所在的位置，它可能靠近多个 Voronoi 像元的边界，这使得从哪个像元返回最近的邻元变得不明确，从而导致边缘问题。因此，IVF-PQ索引涉及设置一个附加参数， `n_probes` 该参数告诉搜索算法向外扩展到 `n_probes` 参数指定的单元格数。

To effectively build an IVF-PQ index, it is necessary to make two choices: the number of subvectors for the PQ step, and the number of partitions for the IVF step. Larger the number of subvectors, the smaller each subspace is, reducing information loss due to compression. However, a larger number of subvectors also results in more I/O and computation within each PQ step, so this number must be minimized to keep computational costs low.

为了有效地构建IVF-PQ索引，需要做出两个选择：PQ步骤的子向量数和IVF步骤的分区数。子向量的数量越多，每个子空间越小，从而减少由于压缩而导致的信息损失。但是，更多的子向量也会导致每个PQ步骤中更多的I / O和计算，因此必须最小化此数字以保持较低的计算成本。

Similarly, the number of partitions in the IVF step must be chosen to balance the trade-off between recall and search speed. The limiting case of number of partitions being equal to the number of vectors in the dataset is *brute force search*, which is the most accurate (recall of 1), but it essentially makes it an IVF-Flat index.

同样，必须选择IVF步骤中的分区数，以平衡召回和搜索速度之间的权衡。分区数等于数据集中向量数的限制情况是暴力搜索，这是最准确的（回想一下 1），但它本质上使其成为 IVF-Flat 索引。

Indeed, the LanceDB IVF-PQ index docs[8] describe exactly these kinds of trade-offs when creating an index, so it's worth going deeper into these concepts to understand how to apply them in a real world scenario.

事实上，LanceDB IVF-PQ 索引文档准确地 [8] 描述了创建索引时的这些权衡，因此值得深入研究这些概念，以了解如何在现实世界场景中应用它们。

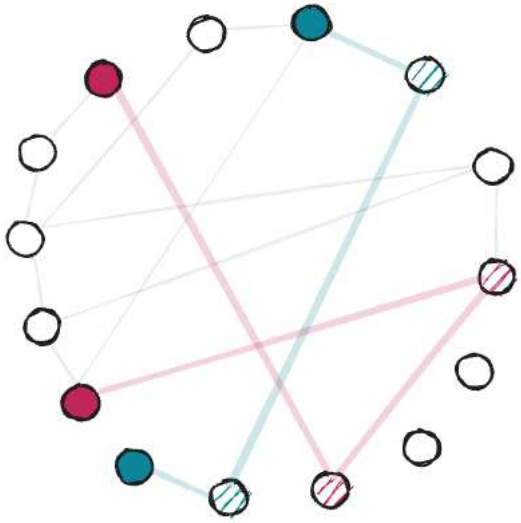
HNSW 高净值

Hierarchical Navigable Small-World (HNSW) graphs is among the most popular algorithms for building vector indexes – as of writing this post, nearly *every* database vendor out there uses it as the primary option. It's also among the most intuitive algorithms out there, and it's highly recommended that you give the original paper that introduced it, a read.

分层可导航小世界（HNSW）图是构建向量索引的最流行的算法之一 - 在撰写本文时，几乎每个数据库供应商都将其用作主要选项。它也是最直观的算法之一，强烈建议您阅读介绍它的原始论文。

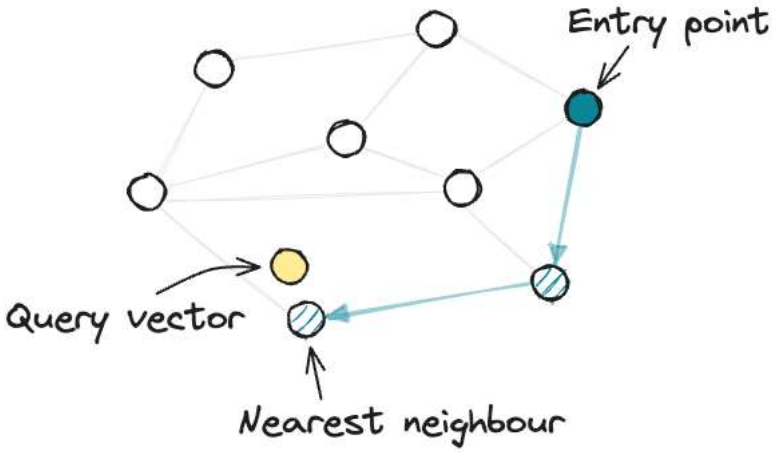
At a high level, HNSW builds on top of the small world graph phenomenon, which states that even though most nodes in a graph are not neighbours of each other, the *neighbours* of any given nodes are likely to be neighbours of each other, regardless of the size of the graph. Essentially, any node in the graph can be reached from every other node by a relatively small number of steps. In the example below, the solid-filled blue and red nodes can be reached from each other in just 3 hops, even though they might be perceived as distant in the graph.

在高层次上，HNSW建立在小世界图现象之上，该现象指出，即使图中的大多数节点不是彼此的邻居，任何给定节点的邻居都可能是彼此的邻居，无论图的大小如何。从本质上讲，图中的任何节点都可以通过相对较少的步骤从其他节点到达。在下面的示例中，只需 3 个跃点即可到达实心填充的蓝色和红色节点，即使它们在图中可能被视为遥远。



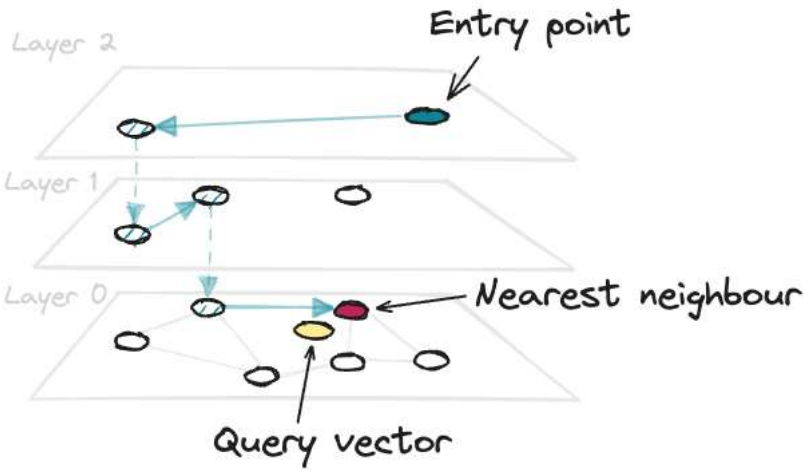
The vector space in which your data lives can also be thought of as a *Navigable* Small World (NSW) graph, where the nodes represent the data points, and edges represent similarities, i.e., numbers that describe how close two nodes are to each other in vector space. NSW works by constructing an undirected graph that ensures global connectivity, i.e., any node can be reached in the graph given an arbitrary entry point. Long edges (connecting nodes that are far apart and require many traversals) are formed first, and short edges (which connect nearby nodes) are formed later on. The long edges improve search *efficiency* and the short edges improve search *accuracy*[3]. The nearest nodes to a given query vector can be found by traversing this graph.

数据所在的向量空间也可以被认为是可导航的小世界 （NSW） 图，其中节点表示数据点，边表示相似性，即描述两个节点在向量空间中彼此接近程度的数字。新南威尔士州的工作原理是构建一个无向图，以确保全球连通性，即在给定任意入口点的情况下，可以在图中到达任何节点。首先形成长边（连接相距很远且需要多次遍历的节点），然后形成短边（连接附近的节点）。长边提高搜索效率，短边提高搜索精度[3]。通过遍历此图，可以找到离给定查询向量最近的节点。



One of the problems with an NSW graph is that it is flat – certain nodes can create dense “traffic hubs”, reducing the efficiency of the traversal and causing the search complexity of the method to be poly-logarithmic[9]. HNSW addresses this issue via a *hierarchical* graph structure and also fixes the upper bound of each node’s number of neighbours, reducing the search complexity to logarithmic[9]. The basic idea is to separate nearest neighbours into layers in the graph based on their distance scale. The long edges in the graph are kept in the top layers (which is the sparsest layer), with each layer below containing edges that are shorter-distance than the layers above it. The lowest layer forms the complete graph, and the search is performed from top to bottom. If all layers are “collapsed” into one another, the HNSW graph is essentially an NSW graph.

NSW图的问题之一是它是扁平的——某些节点可以创建密集的“流量枢纽”，降低遍历的效率，并导致方法的搜索复杂性是多对数[9]的。HNSW通过分层图结构解决了这个问题，并且还修复了每个节点邻居数量的上限，将搜索复杂性降低到对数[9]。基本思想是根据距离尺度将最近邻划分为图形中的图层。图中的长边保留在顶层（这是最稀疏的层）中，下面的每层都包含比其上方层更短距离的边。最低层形成完整的图形，搜索从上到下执行。如果所有图层都相互“折叠”，则 HNSW 图本质上是新南威尔士州图。



The image above shows how, given an arbitrary entry point at the top layer, it's possible to rapidly traverse across the graph, dropping one layer at a time, until the nearest neighbour to the query vector is found.

上图显示了在顶层给定任意入口点的情况下，如何快速遍历图形，一次丢弃一个图层，直到找到离查询向量最近的邻居。

The biggest strength of HNSW over IVF is that it is able to find approximate nearest neighbours in complex, high-dimensional vector space with a high degree of recall. In fact, at the time of its release ~2019, it produced state-of-the-art results on benchmark datasets specifically with regard to improving recall while also being fast, explaining its immense popularity. However, it is not as memory efficient, unless it is combined with methods like PQ to compress the vectors at search time. Databases like Qdrant and Weaviate, typically implement composite indexes that involve quantization, like HNSW-PQ[10] for these reasons, while also offering tuning knobs to adjust the trade-off between recall and query latency.

HNSW相对于IVF的最大优势在于它能够在复杂的高维向量空间中找到具有高度召回性的近似最近邻。事实上，在发布~2019时，它在基准数据集上产生了最先进的结果，特别是在提高召回率的同时又快速，解释了它的巨大受欢迎程度。但是，它的内存效率不高，除非它与PQ等方法结合使用以在搜索时压缩矢量。像Qdrant和Weaviate这样的数据库通常实现涉及量化的复合索引，如HNSW-PQ [10]，同时还提供调谐旋钮来调整召回和查询延迟之间的权衡。

Vamana 瓦马纳

Vamana is among the most recently developed graph-based indexing algorithms, first presented at NeurIPS 2019 by Subramanya et al. in collaboration with Microsoft Research India.

Vamana是最近开发的基于图形的索引算法之一，由Subramanya等人与Microsoft Research India合作在NeurIPS 2019上首次提出。

The standout features of Vamana are:

Vamana 的突出特点是：

- It was designed from the ground up to work both in-memory (which most indexes are designed to do) as well as *on-disk*, whose implementation as presented by Microsoft, is termed DiskANN.

  它从头开始设计为在内存中（大多数索引旨在这样做）和磁盘上工作，Microsoft 提出的实现称为 DiskANN。

  - On-disk indexes seem to be proving a huge implementation challenge for vector database vendors, so this is a key feature of Vamana that differentiates it from other algorithms

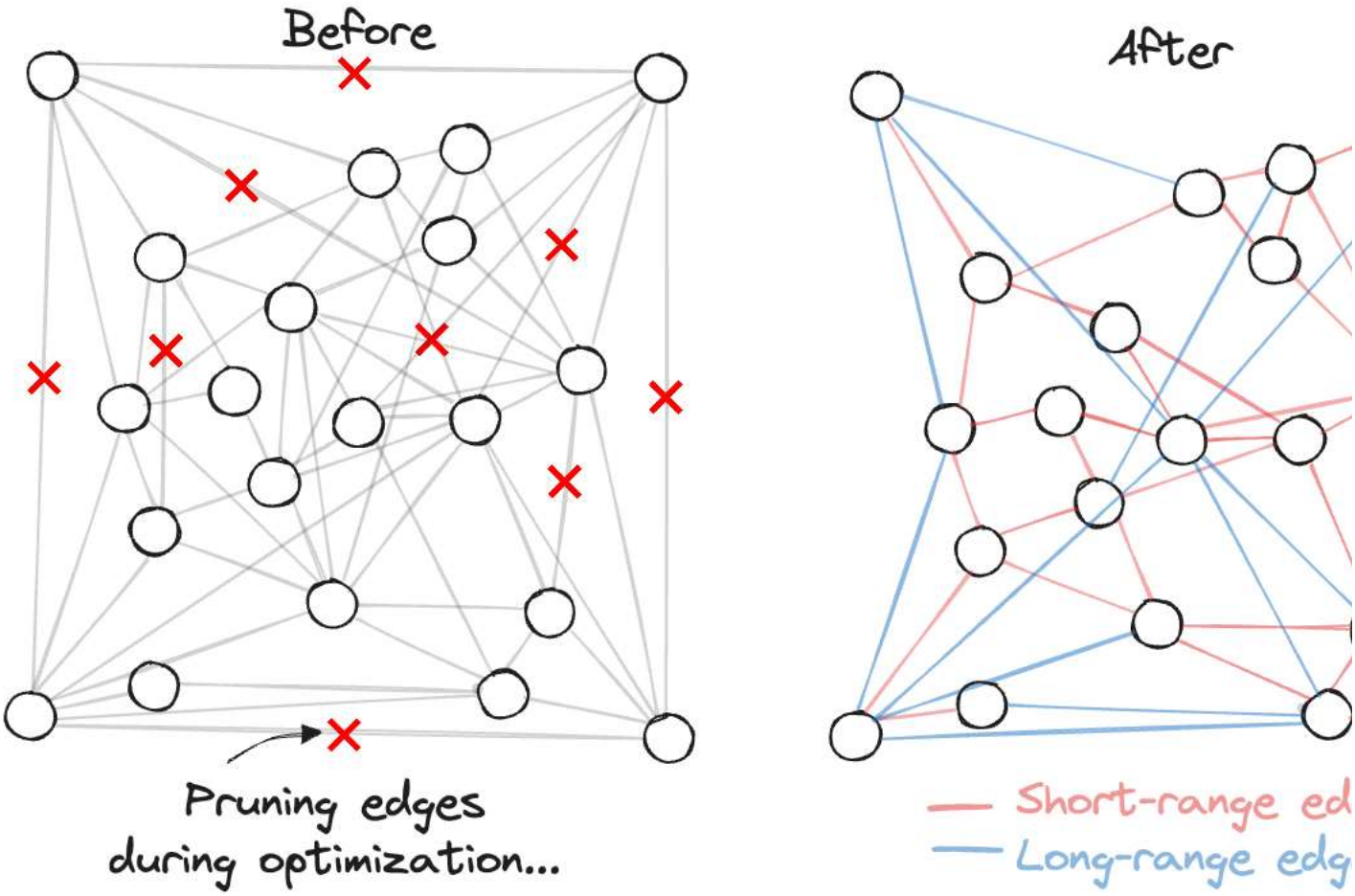
    磁盘索引似乎对矢量数据库供应商来说是一个巨大的实现挑战，因此这是Vamana与其他算法区分开来的关键功能。
- It allows the indexing of datasets that are too large to fit in memory by constructing smaller indexes for overlapping partitions, that can be easily merged into one single index whose query performance is on par with single indexes constructed for the entire dataset

  它允许通过为重叠分区构建较小的索引来索引太大而无法放入内存的数据集，这些索引可以轻松地合并到一个索引中，其查询性能与为整个数据集构建的单个索引相当
- It can also be combined with off-the-shelf vector compression methods like PQ, to build a Vamana-PQ index that powers a DiskANN system – the graph index with the full-precision vectors of the dataset are stored on disk, whereas the compressed vectors are cached in memory, achieving the best of both worlds

  它还可以与PQ等现成的向量压缩方法结合使用，以构建为DiskANN系统提供支持的Vamana-PQ索引 - 具有数据集全精度向量的图形索引存储在磁盘上，而压缩向量缓存在内存中，实现两全其美

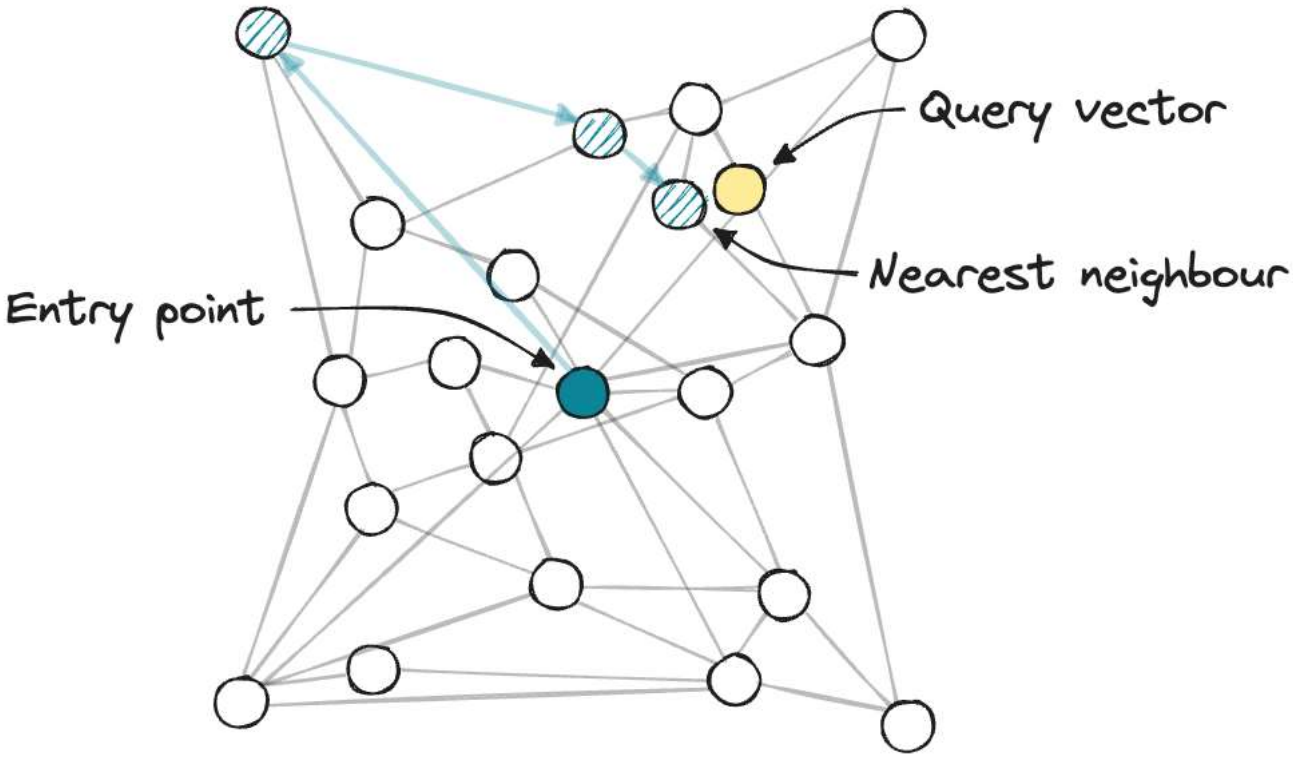
Vamana iteratively constructs a directed graph, first starting off with a random graph, where each node represents a data point in vector space. At the beginning, the graph is well-connected, meaning nearly all nodes are connected to one another. The graph is then optimized using an objective function that aims to maximize the connectivity between nodes that are closest to one another. This is done by pruning most of the random short-range edges, while also adding certain long-range edges that connect nodes that are quite distant from one another (to speed up traversals in the graph).

Vamana 迭代构造一个有向图，首先从一个随机图开始，其中每个节点代表向量空间中的一个数据点。一开始，图形连接良好，这意味着几乎所有节点都相互连接。然后使用目标函数对图进行优化，该函数旨在最大化彼此最接近的节点之间的连接性。这是通过修剪大多数随机短程边来完成的，同时还添加某些连接彼此相距很远的节点的长程边（以加快图中的遍历）。



During query time, the entry point is chosen to be the global centroid. The search rapidly progresses in the right direction via the long-range edges, which allow the algorithm to jump to the ends of the graph and narrow down on the nearest neighbour of the query vector relatively quickly. In the example below, it takes just three hops to traverse from the entry point, which is the global centroid, to the outer edge of the graph, and then to the nearest neighbour.

在查询期间，选择入口点作为全局质心。搜索通过长程边在正确的方向上快速进行，这使得算法能够跳到图的末端，并相对较快地缩小查询向量的最近邻居的范围。在下面的示例中，从入口点（即全局质心）遍历到图形的外边缘，然后到最近的邻居，只需三个跃点。



As you might have observed already, Vamana offers more of an “inside-out” approach to search, as opposed to the “outside-in” approach of HNSW, where the search starts from a random (potentially far out) node in the top layer and progresses inwards.

正如您可能已经观察到的那样，Vamana 提供了更多的“由内而外”的搜索方法，而不是 HNSW 的“由外而内”方法，其中搜索从顶层的随机（可能很远）节点开始并向内进行。

There are not many databases that currently (as of 2023) implement the Vamana index, presumably due to the technical challenges with on-disk implementations and their implications on latency and search speed. Milvus[11], for now, is the only vendor that has a working, on-disk Vamana index, whereas Weaviate[12] and LanceDB[13] currently only have experimental implementations. However, this is a rapidly evolving space, so it’s highly recommended that you follow up on the key vector DB vendors to stay up to date on the latest developments!

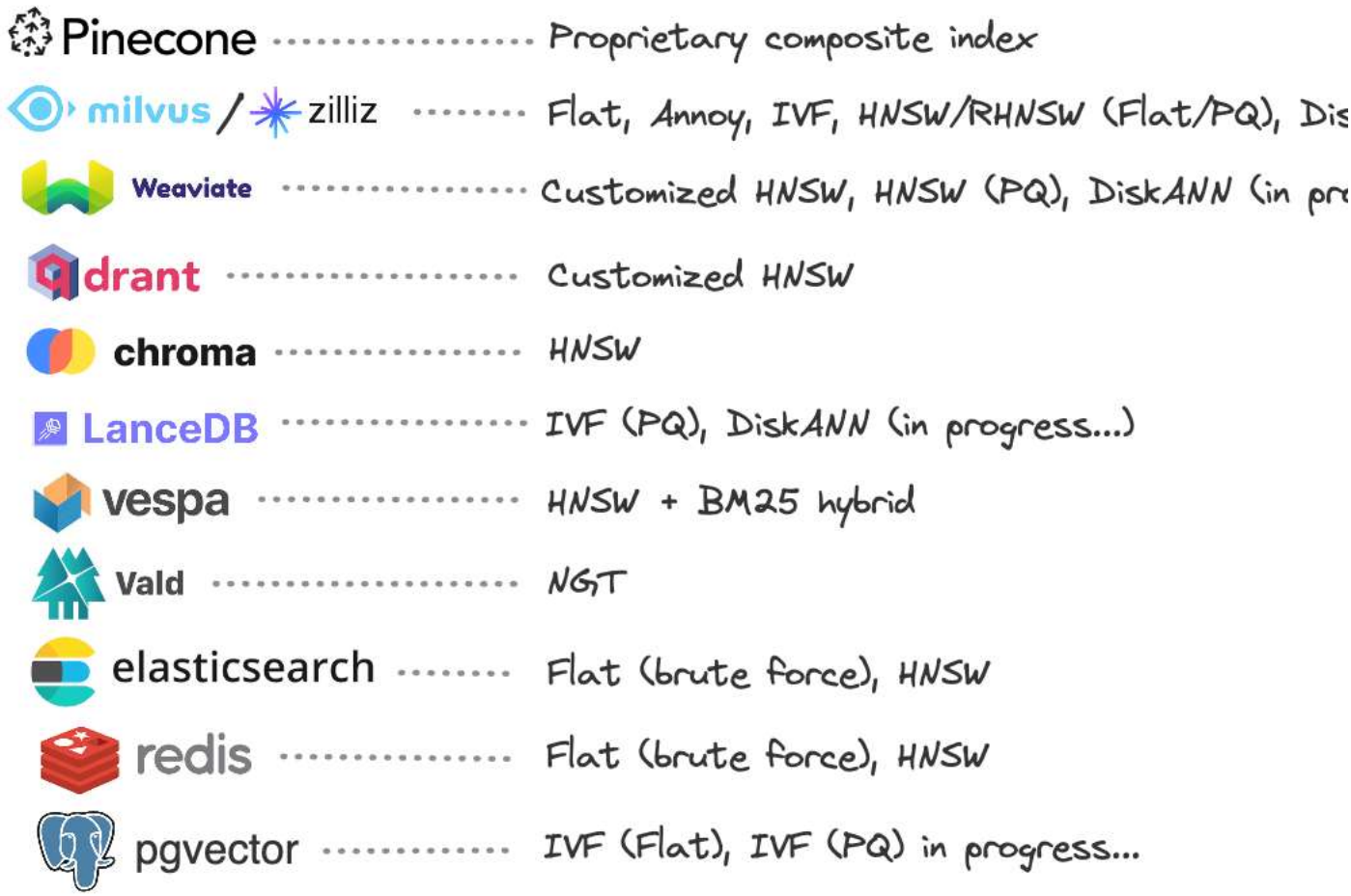
目前（截至 2023 年）实施 Vamana 索引的数据库并不多，这可能是由于磁盘实现的技术挑战及其对延迟和搜索速度的影响。目前，Milvus [11] 是唯一拥有工作磁盘 Vamana 索引的供应商，而 Weaviate [12] 和 LanceDB [13] 目前只有实验性实现。但是，这是一个快速发展的领域，因此强烈建议您跟进关键的向量数据库供应商，以了解最新的发展！

Available indexes in popular vector databases

常用矢量数据库中的可用索引

As shown in part 1 of this series, most databases implement the HNSW index as the default option.

如本系列的第 1 部分所示，大多数数据库将 HNSW 索引实现为默认选项。



Databases like Milvus, Weaviate, Qdrant and LanceDB offer simple tuning knobs to control the compression/quantization levels in their Product Quantization components. It’s important to understand the fundamentals of how these indexes work, so that you can make the right choice of database and indexing parameters for your use case.

像Milvus，Weaviate，Qdrant和LanceDB这样的数据库提供了简单的调谐旋钮来控制其产品量化组件中的压缩/量化级别。了解这些索引工作原理的基础知识非常重要，这样您就可以为您的用例正确选择数据库和索引参数。

Conclusions 结论

The makers of purpose-built vector databases have spent thousands of man hours fine-tuning and optimizing their indexes and storage layers, so if you have large datasets and require < 100 ms latency on vector search queries, resorting to full-fledged, scalable open-source databases like Weaviate, Qdrant and Milvus seems like a no-brainer,

both from a developer's and a business's perspective.

专用矢量数据库的制造商花费了数千个工时来微调和优化其索引和存储层，因此，如果您拥有大型数据集并且需要<矢量搜索查询延迟 100 毫秒，那么从开发人员和企业的角度来看，求助于 Weaviate、Qdrant 和 Milvus 等成熟的、可扩展的开源数据库似乎都是不费吹灰之力的。

- A `Flat` index is one that stores vectors in their unmodified form, and is used for exact kNN search. It is the most accurate, but also the slowest.

  索引是以未修改的形式存储向量的 `Flat` 索引，用于精确的 kNN 搜索。它是最准确的，但也是最慢的。
- `IVF-Flat` indexes use inverted file indexes to rapidly narrow down on the search space, which are much faster than brute force search, but they sacrifice some accuracy in the form of recall

  `IVF-Flat` 索引使用倒排文件索引来快速缩小搜索空间，这比暴力搜索快得多，但它们以调用的形式牺牲了一些准确性
- `IVF-PQ` uses IVF in combination with Product Quantization to compress the vectors, reducing the memory footprint and speeding up search, while being better in recall than a pure `PQ` index

  `IVF-PQ` 将IVF与产品量化结合使用来压缩载体，减少内存占用并加快搜索速度，同时比纯 `PQ` 索引更好的召回率
- `HNSW` is by far the most popular index, and is often combined with Product Quantization, in the form of `HNSW-PQ`, to improve search speed and memory efficiency compared to `IVF-PQ`

  `HNSW` 是迄今为止最受欢迎的索引，并且通常以 的形式 `HNSW-PQ` 与产品量化结合使用，以提高搜索速度和内存效率 `IVF-PQ`
- `Vamana` is a relatively new index, designed and optimized for on-disk performance – it offers the promise of storing larger-than-memory vector data while performing as well, and as fast, as `HNSW`

  `Vamana` 是一个相对较新的索引，针对磁盘性能进行了设计和优化 - 它提供了存储大于内存的矢量数据的承诺，同时性能也一样快，就像 `HNSW`
  - However, it's still early days and not many databases have made the leap towards implementing it due to the challenges of on-disk performance

    但是，由于磁盘性能的挑战，现在仍处于早期阶段，没有多少数据库能够实现它。

In my view, however, LanceDB is among the most exciting databases to watch out for in the coming months. This is because it is the **only** vector database in the market where *all vector indexes are disk-based*. This is because they are innovating on multiple fronts all at once:

然而，在我看来，LanceDB是未来几个月最值得关注的数据库之一。这是因为它是市场上唯一一个所有矢量索引都基于磁盘的矢量数据库。这是因为他们同时在多个方面进行创新：

1. Building a new, efficient columnar data format, Lance, that is aimed at becoming a modern successor to parquet, while also being optimized for vector search

   构建一种新的、高效的列状数据格式 Lance，旨在成为镶木地板的现代继承者，同时还针对矢量搜索进行了优化
   - It's because of this highly efficient storage layer that LanceDB is able to proceed with such confidence on the disk-based indexing front, unlike other vendors

     正是由于这种高效的存储层，LanceDB能够在基于磁盘的索引方面充满信心地进行，与其他供应商不同。
2. Embedded (serverless), purpose-built architecture built from the ground up

   嵌入式（无服务器）、从头开始构建的专用架构
3. Zero-copy data access, which is a huge performance boost for disk-based indexes

   零拷贝数据访问，这是基于磁盘的索引的巨大性能提升
4. Automatic versioning of data without needing additional infrastructure

   自动对数据进行版本控制，无需额外的基础架构
5. Direct integrations with cloud storage providers like AWS S3 and Azure Blob Storage, making it very easy to integrate with existing data pipelines

   与 AWS S3 和 Azure Blob 存储等云存储提供商直接集成，使其非常容易与现有数据管道集成

Regardless of which database you choose for your use case, we can all agree that this is a great time to be alive and experimenting with these tools. I know there was a lot of information in this article, but the references below really helped me understand the internals of vector databases and how they are built from the ground up. I hope you found this post interesting, and let's keep learning! 🚀

无论您为用例选择哪个数据库，我们都同意这是活着并尝试使用这些工具的好时机。我知道这篇文章中有很多信息，但下面的参考资料确实帮助我理解了矢量数据库的内部结构以及它们是如何从头开始构建的。我希望你觉得这篇文章很有趣，让我们继续学习！ 🚀

Other posts in this series

本系列中的其他帖子

- Vector databases (Part 1): What makes each one different?

  矢量数据库（第 1 部分）：是什么让每个数据库与众不同？
- Vector databases (Part 2): Understanding their internals

  矢量数据库（第 2 部分）：了解其内部结构
- Vector databases (Part 4): Analyzing the trade-offs

  向量数据库（第 4 部分）：分析权衡取舍

---

1. Not All Vector Databases Are Made Equal, Dmitry Kan on Medium ↩

   并非所有的矢量数据库都是平等的，德米特里·坎在中等 ↩

2. Trillion-scale similarity, Milvus blog ↩

   万亿级相似性，米尔维斯博客 ↩

3. A Comprehensive Survey and Experimental Comparison of Graph-Based Approximate Nearest Neighbor Search, arxiv.org ↩ ↩

   基于图的近似最近邻搜索的综合调查和实验比较，arxiv.org ↩ ↩

4. Choosing the right vector index, Frank Liu on Substack ↩ ↩

   选择正确的向量索引，弗兰克·刘谈子栈 ↩↩

5. Product quantization for nearest neighbor search, J'egou, Douze & Schmid ↩

   用于最近邻搜索的产品量化，J'egou，Douze & Schmid ↩

6. Scalar Quantization and Product Quantization, Frank Liu on Zilliz blog ↩

   标量量化和乘积量化，Frank Liu 在 Zilliz 博客↩上

7. Product quantization: Compressing high-dimensional vectors by 97%, Pinecone blog ↩

   产品量化：将高维向量压缩 97%，松果博客 ↩

8. ANN indexes, LanceDB docs) ↩

   ANN 索引，LanceDB 文档） ↩

9. Hierarchical Navigable Small-World graphs paper, Malkov & Yashunin ↩ ↩

   分层导航小世界图形论文，马尔科夫和亚顺宁 ↩ ↩

10. HNSW + PQ, Weaviate blog ↩

    HNSW + PQ， Weaviate blog ↩

11. On-disk index, Milvus docs ↩

    磁盘索引，米尔维斯文档 ↩

12. Vamana vs. HNSW, Weaviate blog ↩

    瓦马纳 vs. HNSW， Weaviate 博客 ↩

13. Types of index, LanceDB docs ↩

    索引类型，LanceDB 文档 ↩