AYI NEDJIMI Consultants

Indexation Vectorielle : Techniques : Guide Complet

Catégorie : Intelligence Artificielle Lecture : 32 min Publié le : 07/12/2025 Auteur : Ayi NEDJIMI

Guide technique complet sur les algorithmes d Indexation Vectorielle : Techniques et Algorithmes. Expert en cybersécurité et intelligence.

Pourquoi l'indexation est-elle nécessaire ?

Lorsque vous travaillez avec des embeddings dans une application IA, la recherche de similarité devient rapidement un goulot d'étranglement. Imaginez une base vectorielle contenant 10 millions de documents représentés par des vecteurs de 1536 dimensions (OpenAI ada-002). Une recherche naïve nécessiterait de calculer la distance entre votre requête et les 10 millions de vecteurs stockés. Guide technique complet sur les algorithmes d Indexation Vectorielle : Techniques et Algorithmes. Expert en cybersécurité et intelligence. Dans un contexte où l'intelligence artificielle transforme les pratiques de cybersécurité, la maîtrise de ia indexation vectorielle techniques devient un avantage stratégique pour les équipes techniques. Nous abordons notamment : introduction à l'indexation vectorielle, hnsw (hierarchical navigable small world) et ivf (inverted file index). Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

Complexité algorithmique : Cette approche dite "brute force" ou k-NN exhaustive a une complexité `O(n × d)`, où `n` est le nombre de vecteurs et `d` la dimensionnalité. Pour notre exemple :

- 10M vecteurs × 1536 dimensions × 4 bytes (float32) = **58.6 GB de données** à parcourir
- Temps de calcul : **3-15 secondes** sur CPU moderne (single-thread)
- Impossible à scaler pour applications temps réel (SLA <100ms)

L'indexation vectorielle résout ce problème en créant des **structures de données spécialisées** qui permettent de trouver les vecteurs les plus similaires en temps logarithmique ou sous-linéaire : `O(log n)` au lieu de `O(n)`. C'est l'équivalent d'un index B-tree pour les bases SQL, mais adapté aux espaces vectoriels haute dimension.

Gain de Performance avec Indexation

Sur 10M vecteurs (1536 dim) :

- **Sans index** (brute force) : 3-15s latence, 58GB RAM minimum
- **Avec HNSW** : 10-50ms latence, 12-20GB RAM (avec compression)
- **Gain** : 100-1000x plus rapide, 3-5x moins de mémoire

Recherche exacte vs approximative (ANN)

Il existe deux approches fondamentales pour la recherche de voisins dans un espace vectoriel :

k-NN Exact (k-Nearest Neighbors Exhaustive)
La recherche k-NN exacte **garantit de trouver les k vecteurs les plus proches** en calculant la distance avec tous les vecteurs de la base. C'est la "vérité terrain" (ground truth) utilisée comme référence pour évaluer les algorithmes approximatifs.

- **Avantages** : Précision 100% (recall@k = 1.0), pas de faux négatifs
- **Inconvénients** : Complexité O(n), impossible à scaler au-delà de 1-10M vecteurs
- **Cas d'usage** : Datasets <100K vecteurs, benchmarks, validation d'algorithmes
- **Implémentations** : FAISS IndexFlatL2, NumPy/SciPy pairwise_distances

ANN (Approximate Nearest Neighbors)
Les algorithmes ANN acceptent un **compromis précision/vitesse** en ne garantissant pas de trouver les k voisins exacts, mais en offrant une très bonne approximation (recall 95-99%+) avec une latence drastiquement réduite.

Notre avis d'expert
Chez Ayi NEDJIMI Consultants, nous constatons que la majorité des organisations sous-estiment les risques liés aux modèles de langage déployés en production. La sécurité des LLM ne se limite pas au prompt engineering : elle exige une approche systémique couvrant les embeddings, les pipelines de données et les mécanismes de contrôle d'accès aux API.

Votre organisation est-elle prête à faire face aux attaques basées sur l'IA ?

- **Avantages** : Complexité O(log n), scalable à des milliards de vecteurs, latence <100ms
- **Inconvénients** : Précision non garantie, tuning des hyperparamètres nécessaire
- **Cas d'usage** : Production ML, RAG, moteurs de recommandation, recherche sémantique
- **Algorithmes** : HNSW, IVF, Product Quantization, LSH (voir sections dédiées)

Le Mythe du 100% Recall

En pratique, un recall de 95-98% est largement suffisant pour la majorité des applications. Les 2-5% de vecteurs manqués ont souvent des scores de similarité marginalement différents et n'impactent pas significativement la qualité du résultat final (ex: dans un RAG, un document avec un score de 0.82 vs 0.81 est fonctionnellement équivalent).

Le compromis vitesse-précision-mémoire

L'indexation vectorielle implique trois dimensions à optimiser simultanément. Il est impossible d'optimiser les trois à leur maximum : vous devez faire des arbitrages selon vos contraintes.

Dimension	Objectif	Leviers d'Optimisation	Trade-off
Vitesse (Latence)	Recherche <50ms P95	Index en RAM, HNSW, GPU acceleration	Coût mémoire élevé, moins de précision
Précision (Recall)	Recall >98%	Augmenter efSearch (HNSW), nprobe (IVF)	Latence augmentée, plus de calculs
Mémoire	Minimiser RAM/ coût	Product Quantization, compression, disk storage	Latence +50-200%, recall réduit 2-5%

Scénarios Typiques d'Arbitrage
Configuration selon Votre Use Case

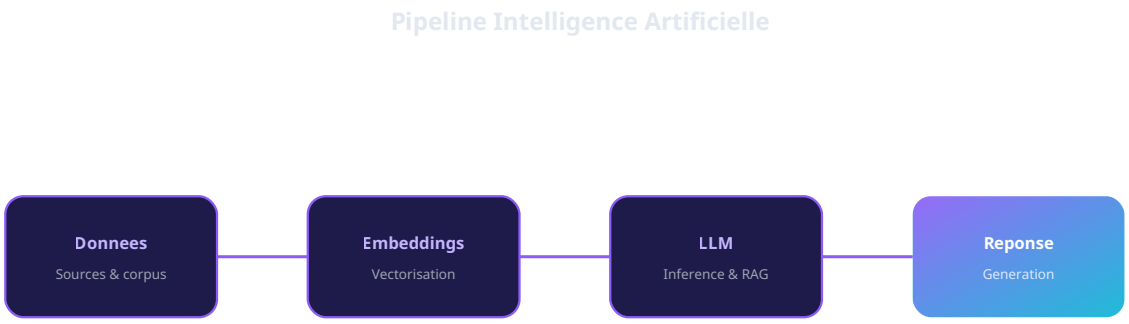
- **Chatbot temps réel (latence critique)** : HNSW in-memory, M=64, efSearch=100 → 20ms, recall 98%, 15GB RAM/10M vecteurs
- **Recommandation produits (coût critique)** : IVF-PQ, 4096 clusters, PQ m=16 → 40ms, recall 95%, 3GB RAM/10M vecteurs
- **Recherche scientifique (précision critique)** : HNSW, M=128, efSearch=500 → 150ms, recall 99.5%, 25GB RAM/10M vecteurs

Vue d'ensemble des familles d'algorithmes

Les algorithmes d'indexation vectorielle se regroupent en quatre grandes familles, chacune avec ses principes, avantages et limites :

Famille	Principe	Algorithmes Clés	Force	Faiblesse
Graph-based	Navigation dans graphe de proximité	HNSW, NSG, DiskANN	Meilleur recall/ latence	Construction lente, RAM
Partitioning	Clustering + recherche locale	IVF, IMI, K-means tree	Scalable, parallélisable	Recall sensible à nprobe
Compression	Quantization des vecteurs	PQ, SQ, OPQ, Residual PQ	Mémoire optimale	Perte de précision
Hashing	Projection aléatoire	LSH, Annoy, MinHash	Construction rapide	Recall inférieur

Dans la pratique moderne (2024-2025), **HNSW domine pour la précision/latence** (utilisé par Qdrant, Weaviate, Vespa), tandis que **IVF+PQ reste le champion de la compression** (FAISS, Milvus en mode économique). LSH et Annoy sont progressivement remplacés par ces solutions plus performantes.



Architecture IA - Du traitement des donnees a la generation de reponses

HNSW (Hierarchical Navigable Small World)

Principe de fonctionnement

HNSW (Hierarchical Navigable Small World Graphs) est l'**algorithme d'indexation vectorielle le plus performant** en termes de rapport recall/latence depuis sa publication en 2016 par Yury Malkov et Dmitry Yashunin. Il combine deux concepts clés :

1. **Navigable Small World (NSW)** : Un graphe où chaque nœud est connecté à ses voisins proches + quelques connexions longue distance, permettant une navigation efficace (inspiration : six degrés de séparation)
2. **Hiérarchie multi-niveaux** : Plusieurs couches de graphes superposées, des plus éparses (en haut) aux plus denses (en bas), à la manière d'un skip-list probabiliste

L'idée centrale : au lieu de parcourir tous les vecteurs, HNSW **navigue dans un graphe** en "sautant" de voisin en voisin, en se rapprochant progressivement de la zone cible. La hiérarchie permet de faire de grands bonds initiaux, puis d'affiner progressivement.

Structure de graphe hiérarchique

HNSW organise les vecteurs en `L` couches (layers) superposées, où chaque couche `l` contient un sous-ensemble des nœuds de la couche `l-1` :

- **Couche 0** (base layer) : Contient TOUS les vecteurs, densément connectés (M*2 connexions par nœud)
- **Couches supérieures** (1, 2, ..., L) : Sous-échantillonnage exponentiel décroissant, connexions longue distance
- **Point d'entrée** : Un nœud unique dans la couche la plus haute, point de départ de toutes les recherches

Exemple de Structure HNSW

Pour 10M vecteurs avec M=16, ml=1/ln(2) ≈ 1.44 :

- **Couche 0** : 10,000,000 vecteurs, ~32 connexions/nœud
- **Couche 1** : ~370,000 vecteurs (3.7%)
- **Couche 2** : ~13,500 vecteurs (0.13%)
- **Couche 3** : ~500 vecteurs
- **Couche 4** : ~18 vecteurs
- **Couche 5** : 1 vecteur (entry point)

Construction de l'index

La construction d'un index HNSW se fait par **insertions séquentielles** des vecteurs. Pour chaque vecteur inséré :

1. **Détermination du niveau** : Tirage aléatoire du niveau max `lc` pour ce nœud avec probabilité exponentielle décroissante : `lc = floor(-ln(uniform(0,1)) * ml)`
2. **Recherche des voisins** : Navigation depuis l'entry point jusqu'à la couche 0 pour trouver les `M` plus proches voisins à chaque niveau ≤ lc
3. **Connexion bidirectionnelle** : Création d'arêtes entre le nouveau nœud et ses M voisins + ajout de l'arête retour
4. **Pruning** : Si un nœud existant dépasse M connexions, suppression des arêtes les plus longues (heuristique de diversité)

Complexité de construction : `O(N * log(N) * M * d)` pour N vecteurs de dimension d. Sur CPU moderne, compter **500-2000 vecteurs/sec** pour des dimensions 768-1536.

Ordre d'Insertion et Qualité de l'Index

L'ordre d'insertion des vecteurs **impacte la qualité finale** de l'index HNSW. L'insertion aléatoire ou stratifiée donne de meilleurs résultats que l'insertion ordonnée (par clusters). Certaines implémentations (Qdrant) offrent une phase de "re-indexation" post-construction pour optimiser le graphe.

Algorithme de recherche

La recherche dans HNSW se déroule en trois phases, de haut en bas des couches :

Phase 1 : Navigation Grossière (Couches Hautes)
Départ depuis l'entry point de la couche supérieure. À chaque étape :

Cas concret
En février 2024, une entreprise de Hong Kong a perdu 25 millions de dollars après qu'un employé a été trompé par un deepfake vidéo lors d'une visioconférence. Les attaquants avaient recréé l'apparence et la voix du directeur financier à l'aide de modèles d'IA générative, démontrant les risques concrets de cette technologie en contexte corporate.

- Évaluation des voisins directs du nœud courant
- Sélection du voisin le plus proche du vecteur de requête

• Si aucun voisin n'est plus proche → descendre d'une couche

Phase 2 : Recherche Locale (Couches Intermédiaires)
Même processus que phase 1, mais avec une liste de candidats étendue (`efSearch` au lieu de 1).

Phase 3 : Affinement Final (Couche 0)
Sur la couche de base, exploration exhaustive de la zone locale avec :

• Priority queue des `efSearch` meilleurs candidats
• Expansion itérative vers les voisins non visités
• Arrêt quand aucun candidat ne peut améliorer les top-k actuels

Complexité de recherche : `O(log(N) * M * d)` en moyenne. Nombre typique de calculs de distance : **500-3000 pour efSearch=100** (vs 10M en brute force).

Paramètres clés (M, efConstruction, efSearch)

HNSW expose trois hyperparamètres principaux qui contrôlent le trade-off précision/latence/mémoire : Pour approfondir, consultez Human-AI Collaboration 2026 : Travailler avec des Agents.

Paramètre	Description	Impact	Valeurs Typiques
M	Nombre de connexions bidirectionnelles par nœud	↑ M : +recall, +latence, + +mémoire ↓ M : -recall, -latence, -- mémoire	Faible dim (128-384) : M=8-16 Moyenne dim (768) : M=16-32 Haute dim (1536) : M=32-64
efConstruction	Taille de la liste de candidats lors de la construction	↑ efC : +qualité index, + +temps construction Généralement efC = 2*M à 4*M	Rapide : efC=100 Équilibré : efC=200 Qualité max : efC=500
efSearch	Taille de la liste de candidats lors de la recherche (query-time)	↑ efS : +recall, +latence Réglable en temps réel sans réindexation	Rapide (recall 90%) : efS=50 Équilibré (recall 95%) : efS=100 Précis (recall 99%) : efS=300-500

Configuration Recommandée pour Production

Comment garantir que vos modèles de machine learning ne deviennent pas des vecteurs d'attaque ?

Dimensionnalité 1024-1536 (ex: OpenAI, Cohere) :

• **M = 32** (équilibre mémoire/précision)
• **efConstruction = 128** (construction raisonnable : ~1000 vec/sec/core)
• **efSearch = 100** (baseline, ajuster selon métriques recall réelles)

Résultats attendus : **Recall@10 > 97%, latence P95 < 30ms** pour 1-10M vecteurs

Avantages et limitations

Avantages de HNSW

- **Meilleur rapport recall/latence** : Systématiquement en tête des benchmarks ANN (ann-benchmarks.com)
- **Robustesse dimensionnelle** : Performant de 128 à 2048 dimensions sans tuning majeur
- **Insertion dynamique** : Support natif des insertions/suppressions (contrairement à LSH statique)
- **Pas de phase d'entraînement** : Contrairement à IVF (clustering k-means), construction incrémentale
- **Parallélisation** : Recherches indépendantes facilement multi-threadées

Limitations de HNSW

- **Consommation mémoire** : 40-60 bytes/vecteur (index) + 4*d bytes (vecteurs bruts). Pour 10M vecteurs 1536-dim : 60GB+ RAM
- **Construction lente** : O(N log N), peut prendre plusieurs heures pour 100M+ vecteurs
- **Pas de compression native** : Doit être combiné avec PQ pour réduire l'empreinte mémoire
- **Sensibilité aux données déséquilibrées** : Clustering fort peut dégrader les performances (curse of dimensionality)
- **Coût de mise à jour** : Les modifications fréquentes peuvent fragmenter le graphe

Cas d'usage recommandés

Utiliser HNSW quand :

- **Précision critique** : Applications où un recall <98% est inacceptable (recherche médicale, juridique)
- **Budget RAM suffisant** : Vous pouvez allouer 50-100GB RAM pour l'index
- **Latence P95 <50ms requise** : Chatbots, recherche interactive temps réel
- **Dataset 1M-100M vecteurs** : Sweet spot de HNSW (au-delà, considérer sharding)
- **Insertions modérées** : <10K nouvelles entrées/jour (sinon, considérer re-indexation périodique)

Exemples d'Implémentations HNSW

- **hnswlib** (Malkov) : Lib C++ référence, bindings Python, la plus rapide
- **Qdrant** : Base vectorielle Rust, HNSW optimisé + filtrage métadonnées
- **Weaviate** : Go, HNSW + GraphQL API, intégrations LLM
- **Vespa** : Java, HNSW + full-text search hybride
- **FAISS** : IndexHNSWFlat, IndexHNSWSQ (avec compression)

IVF (Inverted File Index)

Concept de partitionnement de l'espace

IVF (Inverted File Index) adopte une approche radicalement différente de HNSW : au lieu de naviguer dans un graphe, **IVF partitionne l'espace vectoriel en régions** (clusters) et effectue la recherche uniquement dans les clusters les plus proches du vecteur de requête.

L'analogie : imaginez une bibliothèque organisée par thèmes (sciences, littérature, histoire...). Pour trouver un livre sur la physique quantique, vous n'avez pas besoin de parcourir TOUTE la bibliothèque : vous allez directement à la section "Sciences" puis cherchez localement. IVF applique ce principe aux vecteurs.

Architecture IVF en 3 Composants

1. **Centroid list** : `nlist` vecteurs centraux (centroids) représentant chaque cluster
2. **Inverted lists** : `nlist` listes contenant les IDs des vecteurs appartenant à chaque cluster
3. **Vector storage** : Stockage des vecteurs originaux (IVF-Flat) ou compressés (IVF-PQ)

Clustering avec k-means

La construction d'un index IVF commence par une **phase d'entraînement** (training) qui partitionne l'espace vectoriel via l'algorithme k-means :

Étape 1 : Entraînement k-means

- **Initialisation** : Sélection aléatoire de `nlist` centroids initiaux (ou k-means++)
- **Itérations** : Assignation de chaque vecteur au centroid le plus proche, puis recalcul des centroids (moyenne des vecteurs assignés)
- **Convergence** : Arrêt quand les centroids bougent de moins de `ε` (typiquement 20-100 itérations)
- **Complexité** : `O(nlist * N * d * iterations)` → peut prendre 10-60 minutes pour 10M vecteurs

Importance de l'Échantillon d'Entraînement

Pour accélérer l'entraînement, on utilise souvent un **sous-échantillon** (ex: 256K vecteurs au lieu de 10M). Cet échantillon doit être **représentatif de la distribution** : échantillonnage aléatoire uniforme ou stratifié. Un mauvais échantillonnage dégrade significativement le recall.

Étape 2 : Assignation des Vecteurs

Une fois les centroids entraînés, chaque vecteur de la base est assigné au cluster de son centroid le plus proche. Cette assignation est stockée dans les inverted lists.

Choix du Nombre de Clusters (nlist)

Le paramètre `nlist` contrôle la granularité du partitionnement. Règle empirique :

- **Formule heuristique** : `nlist ≈ sqrt(N)` où N = nombre total de vecteurs
- 1M vecteurs → nlist = 1,000
- 10M vecteurs → nlist = 4,096 (puissance de 2 pour optimisations)
- 100M vecteurs → nlist = 16,384

Processus de recherche

La recherche IVF se déroule en trois étapes distinctes :

Phase 1 : Quantization Grossière (Coarse Quantization)
Calcul des distances entre le vecteur de requête et les `nlist` centroids, puis sélection des `nprobe` clusters les plus proches. Cette phase est un k-NN exhaustif, mais sur seulement nlist vecteurs (ex: 4096 au lieu de 10M).

Phase 2 : Recherche Locale dans Clusters
Pour chaque cluster sélectionné, calcul de la distance entre le vecteur de requête et tous les vecteurs de ce cluster. Les vecteurs sont chargés depuis le storage (RAM ou disque selon implémentation).

Phase 3 : Agrégation et Reranking
Fusion des résultats des `nprobe` clusters, tri par distance, et retour des top-k.

Exemple de Recherche IVF

Configuration : 10M vecteurs, nlist=4096, nprobe=32

- **Phase 1** : Calcul distance avec 4,096 centroids (~0.5ms)
- **Phase 2** : Recherche dans 32 clusters, ~78K vecteurs total (10M/4096*32) (~5-15ms)
- **Phase 3** : Tri et sélection top-10 (~0.1ms)
- **Latence totale** : ~6-16ms vs 3-15s en brute force

IVF-Flat vs IVF-PQ

IVF existe en deux variantes principales, selon le stockage des vecteurs :

Caractéristique	IVF-Flat	IVF-PQ
Stockage vecteurs	Vecteurs bruts (float32)	Vecteurs compressés (Product Quantization)
Mémoire (10M vec 1024-dim)	~40 GB	~2-5 GB (compression 8-20x)
Précision	Haute (identique à brute force dans cluster)	Moyenne (recall -2 à -5% vs IVF-Flat)
Latence	Baseline	15-30% plus rapide (calculs sur codes compressés)
Cas d'usage	Précision max, budget RAM OK	Très gros datasets, contrainte mémoire

Combinaison optimale : IVF-PQ est le choix par défaut pour datasets >10M vecteurs. La compression PQ (voir section dédiée) réduit drastiquement les besoins mémoire avec une perte de recall acceptable (95-98% vs 98-99% pour IVF-Flat).

Paramètre nprobe et optimisation

Le paramètre `nprobe` contrôle le nombre de clusters explorés lors de la recherche. C'est le **levier principal du trade-off recall/latence** pour IVF.

nprobe	% Espace Exploré	Recall Typique	Latence Relative	Use Case
1	0.024% (1/4096)	70-85%	1x (baseline, ~3ms)	Prototypage rapide
8	0.2%	88-93%	~5x	Recommandations approximatives
32	0.78%	95-97%	~15x	**Équilibre production**
128	3.1%	98-99%	~50x	Précision critique
nlist (all)	100%	100%	~1000x	Brute force (debugging)

Stratégie d'Optimisation nprobe

1. **Baseline** : Démarrer avec nprobe = nlist / 128 (ex: 32 pour nlist=4096)
2. **Benchmark** : Mesurer recall@k et latence P95 sur un sample représentatif de requêtes
3. **Ajustement** : Si recall < target, doubler nprobe. Si latence > SLA, réduire de moitié
4. **Monitoring** : En production, tracker les métriques et ajuster dynamiquement selon le trafic

Avantages et limitations

Avantages d'IVF

- **Scalabilité massive** : Performant jusqu'à plusieurs milliards de vecteurs (avec IVF-PQ)
- **Efficacité mémoire** : Avec PQ, 10-30x moins de RAM que HNSW
- **Parallélisation naturelle** : Recherche dans clusters indépendants = GPU-friendly
- **Coût prévisible** : Latence proportionnelle à nprobe (contrairement à HNSW où worst-case difficile à borner)
- **Insertion batch efficace** : Ajout de millions de vecteurs sans dégradation (contrairement à HNSW)

Limitations d'IVF

- **Phase d'entraînement** : Clustering k-means prend 10-60 min (vs construction incrémentale HNSW)
- **Recall inférieur à HNSW** : À latence égale, recall typiquement -2 à -5% vs HNSW
- **Sensibilité à la distribution** : Clusters déséquilibrés (certains avec 100x plus de vecteurs) dégradent les performances
- **Mise à jour coûteuse** : Modifications importantes nécessitent re-training complet des centroids
- **Curse of dimensionality** : Efficacité réduite au-delà de 2048 dimensions (centroids trop proches)

Cas d'usage recommandés

Utiliser IVF quand :

- **Dataset massif** : >100M vecteurs où HNSW devient prohibitif en mémoire
- **Contrainte budgétaire** : Coût RAM/GPU est le facteur limitant (IVF-PQ = 5-10x moins cher)
- **Recall 95% acceptable** : Applications où 3-5% de perte vs HNSW est tolérable
- **Batch processing** : Insertions massives périodiques plutôt que temps réel
- **GPU disponible** : FAISS-GPU accélère IVF de 10-50x (HNSW moins GPU-friendly)
- **Recherche hybride** : Combinaison full-text + vectorielle (Elasticsearch + IVF)

Exemples d'Implémentations IVF

- **FAISS** (Meta) : IndexIVFFlat, IndexIVFPQ, optimisations GPU, référence du domaine
- **Milvus** : IVF-SQ8, IVF-PQ, distribution avec sharding auto
- **Elasticsearch** : IVF avec dense_vector (depuis v8.0)
- **ScaNN** (Google) : Variante optimisée IVF avec anisotropic quantization

PQ (Product Quantization)

Principe de compression vectorielle

Product Quantization (PQ) n'est pas un algorithme d'indexation à proprement parler, mais une **technique de compression** qui réduit drastiquement l'empreinte mémoire des vecteurs en les encodant avec des codes compacts. PQ est souvent combiné avec IVF (IVF-PQ) ou HNSW pour permettre l'indexation de milliards de vecteurs.

L'idée centrale : au lieu de stocker un vecteur de 1536 dimensions en float32 (6144 bytes), PQ le représente avec un **code de 16-64 bytes**, soit une compression de 50-400x, tout en préservant suffisamment d'information pour calculer des distances approximatives.

Exemple de Gain Mémoire avec PQ

Dataset : 100 millions de vecteurs OpenAI ada-002 (1536 dim) Pour approfondir, consultez IA Générative pour le Pentest Automatisé : Méthodes et Outils.

- **Sans compression** : 100M × 1536 × 4 bytes = **614 GB**
- **Avec PQ m=64, k=256** : 100M × 64 bytes = **6.4 GB** (compression 96x)
- **+ Codebooks** : 64 × 256 × 24 bytes = 393 KB (négligeable)
- **Total** : ~6.4 GB vs 614 GB = passage de 14 serveurs 64GB à un seul !

Décomposition en sous-espaces

PQ fonctionne en décomposant chaque vecteur haute dimension en `m` sous-vecteurs de dimension `d/m`, puis en quantifiant indépendamment chaque sous-vecteur.

Étape 1 : Partitionnement du Vecteur
Un vecteur `x ∈ ℝ^d` est découpé en `m` segments contigus :

- **Exemple** : Vecteur 1536-dim avec m=64 → 64 sous-vecteurs de 24 dimensions chacun
- `x = [x₁, x₂, ..., x₆₄]` où chaque `xᵢ ∈ ℝ²⁴`
- Le découpage est fixe et identique pour tous les vecteurs

Optimized Product Quantization (OPQ)

Le découpage contigu naïf de PQ n'est pas optimal car les dimensions voisines peuvent être corrélées. **OPQ** (Optimized Product Quantization) apprend une rotation de l'espace vectoriel avant découpage pour minimiser les corrélations entre sous-espaces. Gain de recall typique : +2-5% pour même taux de compression.

Codebooks et quantification

Pour chaque sous-espace, PQ entraîne un **codebook** (dictionnaire) via k-means contenant `k` centroids (typiquement k=256 pour encoding sur 8 bits = 1 byte).

Phase d'Entraînement

1. **Découpage** : Tous les vecteurs du training set sont découpés en m sous-vecteurs
2. **Clustering indépendant** : Pour chaque sous-espace i (i=1..m), exécution de k-means sur tous les sous-vecteurs x_i pour obtenir k=256 centroids
3. **Stockage codebooks** : Résultat = m codebooks de k centroids chacun (ex: 64 × 256 × 24 floats)

Phase d'Encodage
Pour encoder un nouveau vecteur x :

1. Découpage en m sous-vecteurs : `x = [x₁, ..., xₘ]`
2. Pour chaque sous-vecteur x_i, trouver le centroid le plus proche dans le codebook i → index $c_i \in [0, 255]$
3. Le vecteur compressé est la concaténation des m indices : `code(x) = [c₁, c₂, ..., cₘ]`
4. Stockage : m bytes au lieu de d × 4 bytes

Configuration PQ	Taille Code (bytes)	Compression (1536-dim)	Recall Typique
m=16, k=256	16 bytes	384x	85-92%
m=32, k=256	32 bytes	192x	90-95%
m=64, k=256	64 bytes	96x	93-97%
m=96, k=256	96 bytes	64x	95-98%

Calcul de distance approximatif

La magie de PQ : il est possible de calculer une **distance approximative** entre deux vecteurs compressés SANS les décompresser, via des lookup tables précalculées.

Asymmetric Distance Computation (ADC)
Cas typique : requête (non compressée) vs base de données (compressée PQ)

1. **Précalcul** : Pour le vecteur de requête q, découper en m sous-vecteurs [q_1, ..., q_m]
2. **Distance tables** : Pour chaque sous-espace i, calculer les distances entre q_i et les k=256 centroids du codebook i → m tables de 256 distances
3. **Lookup** : Pour un vecteur compressé code(x)=[c_1,...,c_m], la distance approximative est : `d(q,x) ≈ √(Σᵢ distance_tableᵢ[cᵢ]²)`
4. **Complexité** : O(m) au lieu de O(d) → gain 16-64x en vitesse de calcul

Performance ADC

Sur CPU moderne, le calcul ADC permet d'évaluer **1-5 millions de distances/seconde** (vs 50-200K pour distances float32 exactes). Combiné avec IVF, cela permet de rechercher dans des milliards de vecteurs en <50ms.

Réduction de la consommation mémoire

Au-delà de la compression des vecteurs, PQ impacte positivement toute la chaîne de traitement :

Bénéfices Cascades de PQ

- **Cache CPU** : Plus de vecteurs tiennent dans L3 cache (30-100MB) → moins de cache misses
- **Bande passante mémoire** : Transfert RAM → CPU 50-100x réduit → goulot d'étranglement éliminé
- **Stockage disque** : Bases 100M+ vecteurs tiennent sur SSD au lieu de RAM → coût 10x inférieur
- **Réseau** : Si recherche distribuée, transfert inter-nœuds accéléré
- **GPU** : Plus de vecteurs dans VRAM (16-80GB) → batch processing plus efficace

Variantes de Quantization

Technique	Principe	Compression	Precision
Scalar Quantization (SQ)	Conversion float32 → int8/uint8	4x	Haute (recall -1%)
Product Quantization (PQ)	Codebooks par sous-espace	16-96x	Moyenne (recall -3 à -8%)
Residual PQ	PQ sur résidu après quantization grossière	32-128x	Moyenne-haute (recall -2 à -5%)
OPQ	PQ avec rotation optimisée	16-96x	Haute (recall +2-5% vs PQ)

Avantages et limitations

Avantages de Product Quantization

- **Compression extrême** : 50-400x réduction mémoire, permet datasets massifs
- **Vitesse de calcul** : ADC 16-64x plus rapide que distance float32
- **Scalabilité coût** : Coût infrastructure réduit de 10-50x pour gros volumes

- **Complémentarité** : S'intègre avec IVF, HNSW, disk-based indexes
- **GPU-friendly** : Lookups vectorisés exploitent massivement les CUDA cores

Limitations de PQ

- **Perte de précision** : Recall -3 à -8% vs vecteurs non compressés
- **Phase d'entraînement** : Clustering k-means sur m sous-espaces = 30-120 min pour 10M vecteurs
- **Sensibilité à la distribution** : Vecteurs très clusterés ou outliers dégradent la quantization
- **Tuning délicat** : Choix de m et k impacte fortement recall et latence, nécessite expérimentation
- **Moins efficace en basse dimension** : PQ optimal pour d ≥ 128, peu utile pour d < 64

Recommandation PQ selon Dataset

- **<1M vecteurs** : PQ inutile, utiliser vecteurs bruts (coût RAM acceptable)
- **1-10M vecteurs** : SQ8 (Scalar Quantization 8-bit) = bon compromis précision/mémoire
- **10-100M vecteurs** : IVF-PQ avec m=32-64, nprobe=32
- **>100M vecteurs** : IVF-OPQ ou Residual PQ, GPU acceleration recommandée

LSH (Locality Sensitive Hashing)

Concept de hashing sensible à la localité

Locality Sensitive Hashing (LSH) adopte une approche radicalement différente des méthodes précédentes : au lieu de structurer l'espace avec des graphes (HNSW) ou des clusters (IVF), LSH utilise des **fonctions de hachage spéciales** qui mappent les vecteurs similaires vers les mêmes buckets (hash codes) avec haute probabilité.

Propriété fondamentale : Si deux vecteurs sont proches selon une métrique de distance (cosine, euclidean), ils ont une **forte probabilité de collision** (même hash). Inversement, des vecteurs éloignés ont une faible probabilité de collision.

Analogie LSH

Imaginez un bar où les gens portent des badges colorés selon leurs intérêts. Les personnes avec des intérêts similaires ont statistiquement plus de chance d'avoir le même badge. Pour trouver des gens qui vous ressemblent, vous cherchez uniquement parmi ceux qui portent votre couleur (bucket) au lieu de parler à tout le monde (brute force).

Formalisation Mathématique
Une famille de fonctions de hachage **H** est dite locality-sensitive pour une distance `d` s'il existe des constantes `r`, `cr`, p_1, p_2 telles que pour toute paire de vecteurs x, y :

- Si `d(x, y) ≤ r` (vecteurs proches) → `P[h(x) = h(y)] ≥` p_1 (haute collision)
- Si `d(x, y) ≥ cr` (vecteurs éloignés, c>1) → `P[h(x) = h(y)] ≤` p_2 (faible collision)
- Où $p_1 > p_2$

Familles de fonctions de hachage

Selon la métrique de distance utilisée, différentes familles LSH existent :

Métrique	Famille LSH	Fonction de Hash	Use Case
Similarité Cosinus	Random Hyperplane	`h(x) = sign(w · x)`	Embeddings normalisés (NLP, images)
Distance Euclidienne	p-stable LSH	`h(x) = floor((w · x + b) / r)`	Vecteurs non-normalisés
Jaccard (ensembles)	MinHash	Min permutation hash	Documents, texte sparse
Hamming (binaire)	Bit sampling	Sélection bits aléatoires	Features binaires

Random projections

Pour la similarité cosinus (cas le plus fréquent pour les embeddings), LSH utilise la méthode des **hyperplans aléatoires** (Random Hyperplane Projection) :

Algorithme Random Hyperplane LSH

1. **Génération des hyperplans** : Tirer `k` vecteurs aléatoires $w_1, ..., w_k$ depuis une distribution gaussienne N(0,1)^d
2. **Projection** : Pour un vecteur x, calculer le produit scalaire avec chaque w_i : $p_i = w_i \cdot x$
3. **Binarisation** : $h_i(x) = 1 \text{ if } p_i \geq 0 \text{ else } 0$
4. **Hash code** : Concaténation des k bits : $hash(x) = [h_1(x), h_2(x), ..., h_k(x)]$ → entier de 0 à 2^k - 1

Intuition géométrique : Chaque hyperplan w_i partitionne l'espace vectoriel en deux demi-espaces (+/-). Avec k hyperplans, l'espace est découpé en 2^k régions. Les vecteurs proches ont forte probabilité d'être dans la même région → même hash.

Probabilité de Collision

Pour deux vecteurs x et y avec angle θ entre eux, la probabilité qu'un hyperplan aléatoire les sépare est $P(collision) = 1 - \theta/\pi$. Exemples :

- θ = 0° (identiques) → P = 100%
- θ = 30° (très similaires) → P = 90.5%
- θ = 60° (similaires) → P = 81.0%
- θ = 90° (orthogonaux) → P = 50%

Amplification avec Tables Multiples
Un seul ensemble de k hyperplans donne un recall faible (~60-80%). Pour l'améliorer, LSH utilise **L tables de hash indépendantes** avec différents hyperplans aléatoires :

- Construction : Générer L ensembles de k hyperplans, calculer L hash codes par vecteur
- Recherche : Hasher la requête avec les L tables, collecter tous les candidats des L buckets

- Trade-off : ↑ L augmente recall mais aussi le nombre de faux positifs (et donc latence)

Configuration typique : k=10-20 bits (1024-1M buckets), L=10-50 tables. Recall@10 = 85-95% selon paramètres.

Multi-probing et optimisations

LSH basique souffre d'un problème : un vecteur proche de la frontière d'un hyperplan peut être hashé dans un bucket adjacent. **Multi-probing LSH** résout cela en explorant plusieurs buckets voisins.

Stratégie Multi-Probing

1. **Hash principal** : Calculer le hash code normal de la requête → bucket principal
2. **Buckets voisins** : Générer des hash codes avec 1-3 bits flipés (distance de Hamming 1-3)
3. **Exploration** : Chercher dans le bucket principal + T buckets voisins (T=5-20)
4. **Gain** : Recall +10-20% pour même nombre de tables L

Autres Optimisations LSH

- **Cross-polytope LSH** : Utilise des polytopes réguliers au lieu d'hyperplans → meilleure distribution
- **Data-dependent LSH** : Apprend les projections depuis les données (au lieu d'aléatoire) → +5-10% recall
- **Spherical LSH** : Optimisé pour vecteurs normalisés (unit sphere)
- **Asymmetric LSH** : Hash asymétrique requête vs base pour réduire faux positifs

Avantages et limitations

Avantages de LSH

- **Construction ultra-rapide** : Pas de k-means ou construction de graphe, juste projection linéaire → 1-5 minutes pour 100M vecteurs
- **Scalabilité théorique** : Complexité sous-linéaire prouvable : O(n^(1/(1+ε)))
- **Insertion dynamique triviale** : Ajout d'un vecteur = calcul de L hash codes et insertion dans buckets (microseconde)
- **Distribution facile** : Tables indépendantes → sharding naturel sur plusieurs machines
- **Empreinte mémoire** : Seulement les hash codes + pointeurs (quelques bytes/vecteur)

Limitations de LSH

- **Recall inférieur** : Typiquement 85-92% vs 95-99% pour HNSW/IVF à latence comparable
- **Tuning complexe** : Choix de k et L est délicat et dataset-dependent
- **Sensibilité dimensionnelle** : Performance se dégrade significativement au-delà de 512-1024 dimensions (curse of dimensionality)
- **Distribution de buckets** : Certains buckets peuvent être sur-peuplés (power law), causant des hotspots
- **Évolution technologique** : Largement supplanté par HNSW/IVF dans les benchmarks récents (2020+)

Statut Actuel de LSH (2025)

LSH était l'état de l'art 2010-2015, mais a été dépassé par HNSW (2016+) et les méthodes basées sur learning (2018+). Aujourd'hui, LSH reste pertinent pour :

- **Streaming data** : Insertions massives temps réel (100K+/sec)
- **Systèmes distribués** : Sharding simple sans coordination
- **Faible latence construction** : Index créé en minutes vs heures pour HNSW
- **Ressources limitées** : Edge computing, mobile (empreinte mémoire minimale)

Pour recherche vectorielle standard, **préférer HNSW ou IVF-PQ**. Pour approfondir, consultez Développement Intelligence Artificielle |.

Autres techniques d'indexation

Annoy (Approximate Nearest Neighbors Oh Yeah)

Développé par Spotify en 2013, **Annoy** est un algorithme basé sur des **arbres de projection aléatoire** (random projection trees). Annoy était très populaire avant l'avènement de HNSW grâce à sa simplicité et sa compatibilité mmap (memory-mapped files).

Principe de Fonctionnement

1. **Construction des arbres** : Création de `n_trees` arbres binaires indépendants (typiquement 10-100)
2. **Partitionnement récursif** : À chaque nœud, choisir 2 points aléatoires, tracer l'hyperplan médiateur, partitionner
3. **Feuilles** : Arrêt quand <K vecteurs dans un nœud (K=100-1000)
4. **Recherche** : Descendre dans chaque arbre jusqu'aux feuilles, agréger candidats, reranker

Caractéristiques

- **Points forts** :
  - Index statique optimisé (mmap = pas de chargement RAM)
  - Construction rapide (plus rapide que HNSW)
  - Empreinte mémoire faible
  - Implémentation simple (1500 lignes C++)
- **Points faibles** :
  - Recall inférieur à HNSW (-5 à -10%)
  - Pas d'insertion dynamique (rebuild complet nécessaire)
  - Performance se dégrade en haute dimension (>512)
- **Use cases** : Systèmes de recommandation musicale (Spotify), index statiques mis à jour quotidiennement/hebdomadairement

Annoy chez Spotify

Spotify utilise Annoy pour indexer des dizaines de millions de tracks et alimenter les fonctionnalités "Discover Weekly" et "Radio". Configuration typique : 50-100 arbres, 100-200ms de latence, recall 90-95%. L'index est reconstruit quotidiennement en batch.

ScaNN (Scalable Nearest Neighbors)

Développé par Google Research en 2019, **ScaNN** combine plusieurs innovations pour atteindre un rapport recall/latence exceptionnellement bon, surpassant même HNSW dans certains benchmarks.

Innovations Clés de ScaNN

- **Anisotropic Vector Quantization** : Quantization qui préserve mieux les distances que PQ standard (+3-5% recall)
- **Two-phase search** :
  1. Phase 1 : Recherche grossière avec quantization agressive (recall 70-80%, très rapide)
  2. Phase 2 : Reranking sur top-N candidats avec vecteurs float32 (recall 98%+)
- **SIMD optimization** : Exploitation intensive des instructions vectorielles (AVX-512) pour calculs de distance
- **Tree + quantization hybrid** : Partitionnement hiérarchique + compression pour le meilleur des deux mondes

Performance

Sur le benchmark GLOVE-100 (1.2M vecteurs 100-dim), ScaNN atteint :

- **Recall@10 = 99.0%** en **0.3ms** (vs HNSW : 98.5% en 0.5ms)
- Empreinte mémoire : 30-50% inférieure à HNSW grâce à quantization
- Construction : Plus rapide que HNSW grâce au tree-building parallélisé

Quand Utiliser ScaNN

- Vous avez besoin du **meilleur recall/latence absolu**
- Infrastructure Google Cloud (TPU/optimisations disponibles)
- Vous pouvez utiliser TensorFlow (ScaNN s'intègre nativement)
- Budget RAM limité mais latence ultra-critique

Limitation : Écosystème moins mature que FAISS/hnswlib, documentation plus limitée

NGT (Neighborhood Graph and Tree)

Développé par Yahoo Japan, **NGT** (Neighborhood Graph and Tree) combine graphes et arbres pour un équilibre construction/recherche.

Architecture Hybride

- **ANNG** (Approximate Nearest Neighbor Graph) : Graphe type HNSW mais avec heuristiques différentes
- **ONNG** (Optimized ANNG) : Variante avec pruning agressif des arêtes pour réduire mémoire
- **QG** (Quantized Graph) : Compression des vecteurs avec quantization
- **Tree-based index** : Alternative plus rapide à construire que graphe

Spécificités

- Optimisé pour CPUs Intel/AMD (pas GPU)
- Très performant sur datasets japonais (embeddings multilingues)

- Insertion dynamique supportée (mieux que Annoy, moins bien que HNSW)
- Documentation principalement en japonais (barrière adoption)

Position dans l'écosystème : NGT est une alternative solide à HNSW/FAISS, particulièrement en Asie, mais moins utilisé en Occident faute de communauté.

DiskANN pour données massives

Développé par Microsoft Research, **DiskANN** résout un problème crucial : comment indexer des **milliards de vecteurs** quand la RAM est insuffisante, en utilisant des SSD NVMe comme extension de mémoire.

Principe : Graph on Disk

1. **Construction** : Création d'un graphe HNSW-like optimisé pour accès SSD (minimiser I/O)
2. **Graph layout** : Organisation des nœuds sur disque pour maximiser locality (voisins proches physiquement)
3. **Compressed vectors** : Vecteurs compressés avec PQ stockés sur SSD
4. **In-memory index** : Petit index en RAM (5-10% du dataset) pour point d'entrée rapide
5. **Beam search** : Navigation dans le graphe avec prefetching I/O agressif

Performance

Métrique	HNSW in-RAM	DiskANN	Note
Dataset supportable	10-100M vecteurs	1-10B vecteurs	DiskANN 10-100x plus scalable
Latence (1B vecteurs)	N/A (out of RAM)	1-5ms (NVMe SSD)	Avec SSD haut de gamme
Coût infrastructure	1TB RAM = $10K+/mois	1TB SSD = $100-500/mois	DiskANN 20-100x moins cher
Recall@10	99%+	97-98%	Légère perte due compression

Use Cases DiskANN

DiskANN est idéal pour :

- **Web-scale search** : Bing (Microsoft) utilise DiskANN pour indexer des milliards de pages
- **E-commerce massif** : Catalogues 100M+ produits avec recherche visuelle
- **Archives scientifiques** : Milliards de papers/molécules/images
- **Contrainte budgétaire** : RAM prohibitive, SSD disponible

Prérequis DiskANN

- **SSD NVMe obligatoire** : SATA SSD trop lent (latence 10-50ms vs 0.1-1ms NVMe)
- **Construction coûteuse** : 10-50h pour 1B vecteurs (vs 2-5h pour HNSW in-RAM)
- **Implémentation complexe** : Code Microsoft moins mature que FAISS/hnswlib

Écosystème DiskANN

- **DiskANN (Microsoft)** : Implémentation C++ open-source originale
- **Milvus 2.3+** : Support DiskANN natif avec distribution
- **Qdrant (roadmap)** : Intégration DiskANN prévue 2025

Comparaison et benchmarks

Méthodologie de benchmark

Évaluer les algorithmes d'indexation vectorielle nécessite une **méthodologie rigoureuse** car les performances varient drastiquement selon le dataset, la dimensionnalité, et les paramètres. Voici le framework standard utilisé par ann-benchmarks.com et la communauté recherche.

Datasets de Référence

Dataset	Taille	Dimension	Type	Use Case
SIFT1M	1M	128	SIFT descriptors (images)	Recherche visuelle, benchmark classique
GLOVE-100	1.2M	100	Word embeddings	NLP, recherche sémantique
Deep1B	1B	96	CNN features (ImageNet)	Scalabilité extrême
OpenAI ada-002	Variable	1536	LLM embeddings	RAG, applications modernes

Métriques Évaluées

- **Recall@k** : Pourcentage des vrais k plus proches voisins retournés (métrique principale)
- **Latence P95/P99** : Temps de réponse 95e/99e percentile (plus réaliste que moyenne)
- **QPS** : Queries per second en multi-threading
- **Memory footprint** : RAM consommée (index + vecteurs originaux si nécessaire)
- **Build time** : Temps de construction index (important pour re-indexation)
- **Update throughput** : Insertions/suppressions par seconde

Protocole Standardisé

1. **Split train/test** : 90% construction index, 10% requêtes test
2. **Tuning paramètres** : Grid search pour optimiser recall@10 = 95% ± 1%
3. **Mesures multiples** : 3-5 runs, report median + std deviation
4. **Hardware fix** : Machine AWS c5.4xlarge (16 vCPU, 32GB RAM) pour reproductibilité
5. **Concurrence réaliste** : Tests mono-thread ET multi-thread (8-16 threads)

Pièges À Éviter dans les Benchmarks

- **Cherry-picking** : Ne tester que sur datasets favorables à votre algorithme
- **Cold vs warm cache** : Mesurer latence après warmup (sinon biais cache misses)
- **Hyperparams non-optimisés** : Utiliser paramètres par défaut sans tuning

- **Hardware non-représentatif** : Benchmark sur 128GB RAM puis déployer sur 16GB

Performance selon la taille du dataset

Les algorithmes d'indexation ont des profils de scalabilité très différents. Voici l'évolution des performances selon la taille N :

Algorithme	100K vecteurs	1M vecteurs	10M vecteurs	100M vecteurs	1B vecteurs
Brute Force	5ms, 100% recall	50ms, 100% recall	5s, 100% recall	50s, 100% recall	8-10min, 100% recall
HNSW	0.5ms, 99% recall	2ms, 98.5% recall	15ms, 98% recall	50-100ms, 97% recall	Impraticable (RAM)
IVF-Flat	1ms, 95% recall	3ms, 95% recall	12ms, 94% recall	40ms, 93% recall	200ms, 92% recall
IVF-PQ	0.8ms, 92% recall	2.5ms, 91% recall	8ms, 90% recall	25ms, 89% recall	80ms, 88% recall
LSH	2ms, 88% recall	4ms, 86% recall	15ms, 83% recall	60ms, 80% recall	300ms, 75% recall
DiskANN	N/A (overhead)	N/A (overhead)	5ms, 97% recall	12ms, 96% recall	30ms, 95% recall

Analyse des Tendances

- **HNSW** : Excellent jusqu'à 10-50M, puis contrainte RAM devient prohibitive
- **IVF-PQ** : Scalabilité linéaire exceptionnelle, choix #1 pour 100M+ vecteurs
- **LSH** : Dégradation continue du recall (curse of dimensionality)
- **DiskANN** : Seule solution viable pour datasets multi-milliards

Précision (recall) vs latence

Le graphique recall vs latence est LA métrique de référence pour comparer algorithmes. Voici les résultats sur SIFT1M (1M vecteurs, 128 dim) :

Champion par Catégorie (SIFT1M)

- **Meilleur recall global** : HNSW (99.2% @ 2.1ms)
- **Latence ultra-faible** : IVF-PQ (91.5% @ 0.8ms)
- **Meilleur équilibre** : HNSW M=32 (98.1% @ 1.5ms)
- **Plus économique** : IVF-PQ (95.5% @ 2ms, 10x moins RAM)

Résultats Détaillés par Algorithme

Config	Recall@10	Latence P95	QPS	Note
HNSW M=16, ef=50	96.8%	1.1ms	4,200	Rapide, recall correct
HNSW M=32, ef=100	98.1%	1.5ms	3,100	Équilibre optimal
HNSW M=64, ef=200	99.2%	2.1ms	2,200	Précision maximale
IVF-Flat n=1024, p=8	93.5%	1.8ms	2,800	Baseline IVF
IVF-PQ m=16, p=16	91.2%	1.2ms	3,500	Compression 32x
LSH k=18, L=20	88.4%	2.5ms	1,800	Construction rapide

Consommation mémoire

L'empreinte mémoire est souvent le facteur limitant en production. Voici la consommation pour 10M vecteurs 1024-dim (float32) :

Algorithme	Index	Vecteurs	Total	Compression
Vecteurs bruts	0 GB	40 GB	40 GB	1x (baseline)
HNSW M=32	15 GB	40 GB	55 GB	0.73x
IVF-Flat	0.5 GB	40 GB	40.5 GB	1.01x
IVF-PQ m=32	0.5 GB	1.25 GB	1.75 GB	23x
LSH k=16, L=50	0.8 GB	40 GB	40.8 GB	0.98x
DiskANN	4 GB (RAM)	1.5 GB (SSD)	5.5 GB	7.3x

Impact Économique de la Compression

Pour 100M vecteurs 1536-dim, passer de HNSW (550GB RAM) à IVF-PQ (15GB RAM) = économie de **$5,000-15,000/mois** en cloud (AWS r5.24xlarge vs r5.xlarge). ROI compression = rentabilité en 1-2 semaines de déploiement.

Temps de construction de l'index

Le temps de construction impacte la fréquence de re-indexation possible et donc la fraîcheur des données. Mesures sur 10M vecteurs 1024-dim (16-core CPU) :

Algorithme	Temps Construction	Parallélisation	Fréquence Max Re-index
HNSW M=32	45-90 minutes	Oui (threads)	1-2x/jour
IVF-Flat	15-30 minutes	Oui (k-means)	4-6x/jour
IVF-PQ	25-45 minutes	Oui	2-4x/jour
LSH	2-8 minutes	Très bien	Temps réel (streaming)
Annoy	10-20 minutes	Partiel	6-12x/jour
DiskANN	3-8 heures	Oui (disque I/O limité)	1x/semaine

Tableau récapitulatif

Synthèse des algorithmes selon 6 critères principaux (note /10) :

Algorithme	Recall	Latence	Mémoire	Scalabilité	Construction	Flexibilité	Total
HNSW	10/10	9/10	4/10	6/10	6/10	8/10	**43/60**
IVF-PQ	7/10	8/10	10/10	10/10	7/10	6/10	**48/60**
LSH	5/10	6/10	7/10	8/10	10/10	9/10	**45/60**
DiskANN	8/10	7/10	9/10	10/10	3/10	4/10	**41/60**
ScaNN	9/10	10/10	8/10	7/10	7/10	5/10	**46/60**

Verdict 2025

- **Champion toutes catégories** : IVF-PQ (48/60) - meilleur équilibre global
- **Précision absolue** : HNSW (43/60) - mais attention à la RAM
- **Innovation prometteuse** : ScaNN (46/60) - surveiller évolution
- **Web-scale** : DiskANN (41/60) - seule option milliards de vecteurs

Optimisation et tuning

Choisir l'algorithme selon vos contraintes

Le choix de l'algorithme d'indexation dépend de votre profil de contraintes. Utilisez cet arbre de décision pour identifier la solution optimale :

Arbre de Décision Algorithmique

1. Quelle est la taille de votre dataset ?

- **< 100K vecteurs** : Brute force ou FAISS IndexFlatL2 (simple et efficace)
- **100K - 1M vecteurs** : HNSW (hnswlib ou Qdrant)
- **1M - 50M vecteurs** : HNSW si budget RAM OK, sinon IVF-PQ
- **50M - 1B vecteurs** : IVF-PQ avec GPU acceleration (FAISS-GPU)
- **> 1B vecteurs** : DiskANN ou Milvus distribué

2. Quel est votre budget RAM disponible ?

- **Budget illimité** : HNSW pour performance maximale
- **Budget serré** : IVF-PQ (compression 20-50x)
- **Très limité** : DiskANN (SSD au lieu de RAM)

3. Quelle latence est acceptable ? Pour approfondir, consultez Optimiser le Chunking de.

- **< 10ms (temps réel)** : HNSW ou ScaNN
- **10-50ms (interactif)** : HNSW ou IVF-PQ bien tuné
- **50-200ms (batch)** : Tous algorithmes, optimiser coût
- **> 200ms** : Focus sur recall et coût, LSH acceptable

Cas d'Usage Types et Recommandations

Scénario	Algorithme Recommandé	Config Suggérée	Justification
Chatbot RAG	HNSW	M=32, ef=100	Latence critique, précision importante
E-commerce reco	IVF-PQ	nlist=4096, m=32, nprobe=16	Millions produits, coût important
Recherche scientifique	HNSW	M=64, ef=300	Recall maximal requis
App mobile	Annoy	n_trees=50	Contrainte taille, mmap friendly
Streaming analytics	LSH	k=16, L=20	Insertions temps réel massives

Tuning des hyperparamètres

L'optimisation des paramètres est cruciale pour atteindre les performances optimales. Voici une méthodologie systématique :

Méthodologie de Tuning

1. **Baseline** : Démarrer avec paramètres par défaut de la documentation
2. **Objectif** : Définir recall@k target (ex: 95%) et latence P95 limite (ex: 50ms)
3. **Grid search** : Tester combinaisons de paramètres sur sample représentatif
4. **Validation** : Mesurer sur dataset complet avec traffic patterns réalistes
5. **Production** : Déployer avec monitoring continu

Guide de Tuning par Algorithme

HNSW - Optimisation Détaillée

Paramètre	Valeur Conservatrice	Valeur Équilibrée	Valeur Agressive	Impact
M	16	32	64	+M = +recall, +RAM, +latence construction
efConstruction	100	200	400	+efC = +qualité index, ++temps construction
efSearch	50	100	300	+efS = +recall, +latence (ajustable runtime)

Règle d'Or HNSW

- **efConstruction** ≥ **M** : Sinon qualité dégradée
- **efSearch** ≥ **k** : où k = nombre de voisins recherchés
- **M optimal** ≈ **dimensionalité / 32** : heuristique pour dimensions 512-2048

IVF-PQ - Optimisation Avancée

- **nlist** : Commencer par sqrt(N), ajuster selon distribution
  - Si clusters déséquilibrés : augmenter nlist
  - Si trop de clusters vides : réduire nlist
- **nprobe** : Équilibre recall/latence
  - Démarrer avec nprobe = nlist / 128
  - Doubler jusqu'à atteindre recall target
- **PQ.m** : Nombre de sous-espaces
  - Contrainte : dimension doit être divisible par m
  - Plus élevé = meilleur recall mais plus de mémoire
  - Sweet spot : m = dimension / 24

Stratégies hybrides

Les systèmes de production combinent souvent plusieurs techniques pour optimiser différents aspects. Voici les patterns les plus efficaces :

Hot/Cold Tiering

Principe : Séparer les données selon leur fréquence d'accès

- **Tier Hot** (20% des données, 80% du trafic) : HNSW en RAM pour latence minimale
- **Tier Warm** (60% des données, 18% du trafic) : IVF-PQ en RAM
- **Tier Cold** (20% des données, 2% du trafic) : DiskANN sur SSD

Implémentation Tiering

Les systèmes comme Pinecone Serverless et Qdrant Cloud implémentent automatiquement ce tiering basé sur des métriques d'accès. Résultat : latence P95 des requêtes hot <10ms, coût global 5-10x inférieur au all-in-RAM.

Multi-Index Serving
Utiliser plusieurs index différents pour le même dataset :

- **Index principal** : IVF-PQ pour 95% des requêtes (latence normale)
- **Index backup** : HNSW pour 5% des requêtes critiques (latence ultra-faible)
- **Routing intelligent** : Dispatcher basé sur priorité requête (temps réel vs batch)

Recherche en Cascade
Exécution séquentielle d'algorithmes de précision croissante :

1. **Étape 1** : IVF-PQ rapide (recall 90%, 5ms) → top-100 candidats
2. **Étape 2** : Reranking HNSW sur top-100 (recall 99%, +3ms) → top-10 final
3. **Bénéfice** : Combiner vitesse IVF + précision HNSW

Monitoring et métriques en production

Un système de recherche vectorielle en production doit être instrumenté pour détecter dégradations et optimiser continuellement.

Métriques Clés à Tracker

Catégorie	Métrique	Seuil Alert	Action
Performance	Latence P95	> SLA + 30%	Ajuster hyperparamètres ou scale up
	QPS soutenu	< 80% capacité théorique	Investiguer goulots, optimiser code
	Recall (si ground truth)	< target - 5%	Retuning paramètres ou re-indexation
Ressources	RAM utilisation	> 85%	Scale up ou compression
	CPU utilisation	> 80% sustained	Scale horizontalement
Qualité	Taux erreur 5xx	> 0.1%	Debug immédiat
	Timeouts	> 1%	Optimiser ou augmenter timeout

Monitoring Avancé : Drift Detection
Les embeddings peuvent dériver avec le temps (concept drift), dégradant la qualité de recherche :

- **Distribution monitoring** : Tracker moyenne/variance des embeddings par batch
- **Cluster drift** : Pour IVF, surveiller l'équilibre des clusters (coefficient Gini)
- **Query patterns** : Détecter changements dans distribution des requêtes
- **Semantic coherence** : Évaluer périodiquement sur jeu de test semantique

Quand réindexer ?

La réindexation est coûteuse mais parfois nécessaire. Voici les triggers et stratégies :

Triggers de Réindexation
Réindexation Obligatoire Quand :

- **Modèle d'embedding changé** : OpenAI ada-002 → ada-003, nouvelle version Cohere
- **Dimensionnalité modifiée** : 1536 → 3072 dimensions
- **Dataset size doublement+** : Index optimisé pour 1M, maintenant 10M vecteurs
- **Dégradation critique** : Recall < 85% ou latence >2x objectif

Stratégies de Réindexation

- **Blue-Green Deployment** :
  - Construire nouvel index en parallèle ("green")
  - Une fois prêt, switcher le trafic atomiquement
  - Conserver ancien index ("blue") pour rollback rapide
- **Incremental Reindexing** :
  - Pour algorithmes supportant updates (HNSW)
  - Réindexer par batches de 10-100K vecteurs
  - Plus complexe mais zéro downtime
- **Sharded Reindexing** :
  - Réindexer un shard à la fois
  - Trafic redirigé sur shards sains temporairement
  - Scalabilité linéaire mais capacité réduite temporairement

Calendrier de Réindexation Recommandé

- **HNSW** : Réindexation complète tous les 3-6 mois (dégradation progressive)
- **IVF-PQ** : Re-training centroids tous les mois, rebuild complet tous les 6 mois
- **LSH** : Génération nouveaux hash aléatoires tous les 1-2 mois
- **DiskANN** : Réindexation seulement si changement majeur (coût élevé)

Sources et références : ArXiv IA · Hugging Face Papers

Questions fréquentes

Quel algorithme d'indexation est le plus rapide ?

La réponse dépend de votre définition de "rapide" :

- **Latence query la plus faible** : ScaNN (Google) ou HNSW bien tuné (<1ms possible)
- **Construction la plus rapide** : LSH (minutes vs heures pour HNSW/IVF)
- **Meilleur QPS** : IVF-PQ sur GPU (100K+ queries/sec avec FAISS-GPU)
- **Insertions temps réel** : LSH (>100K insertions/sec vs 1K/sec pour HNSW)

En pratique, **IVF-PQ est souvent le plus "rapide" globalement** car il offre le meilleur compromis vitesse de construction, latence query, et scalabilité.

Comment mesurer la qualité d'un index ?

Trois métriques principales :

- **Recall@k** : Pourcentage des vrais k plus proches voisins retournés. Métrique de référence, objectif typique : recall@10 > 95%
- **Precision@k** : Moins utilisée pour ANN (toujours 100% par définition)
- **Mean Average Precision (MAP)** : Pour évaluation plus fine, pondère l'ordre des résultats

Méthode de calcul : Comparer résultats algorithme ANN vs ground truth (k-NN exhaustif) sur un sample de 1K-10K requêtes représentatives. Outils : FAISS benchmark, ann-benchmarks.com

Peut-on combiner plusieurs techniques d'indexation ?

Absolument, et c'est même recommandé en production ! Stratégies courantes :

- **IVF + PQ** : Standard, combine partitionnement et compression
- **HNSW + Scalar Quantization** : Précision HNSW avec 4x moins de RAM
- **Two-phase search** : IVF-PQ pour screening + HNSW pour reranking final
- **Multi-index serving** : HNSW pour requêtes critiques, IVF-PQ pour le reste
- **Hierarchical clustering** : IVF global + HNSW par cluster

Le système Pinecone utilise par exemple une combinaison IVF + PQ + filtrage métadonnées optimisée.

L'indexation fonctionne-t-elle différemment selon la dimension des vecteurs ?

Oui, l'efficacité des algorithmes varie drastiquement avec la dimensionnalité :

- **Basse dimension (<64)** : Tous algorithmes fonctionnent bien, brute force souvent suffisant
- **Dimension moyenne (64-512)** : Sweet spot pour la plupart des algorithmes
- **Haute dimension (512-2048)** : HNSW et ScaNN excellent, LSH se dégrade
- **Très haute dimension (>2048)** : Curse of dimensionality, considérer réduction dimensionnelle (PCA, UMAP)

Règle empirique : Paramètre M de HNSW = dimension/32, nombre de clusters IVF = sqrt(N) × (1 + dimension/1000)

Comment gérer les mises à jour fréquentes de l'index ?

Plusieurs stratégies selon votre fréquence de mise à jour :

Pour approfondir, consultez les ressources officielles : Hugging Face, arXiv et ANSSI.

- **<1K updates/jour** : HNSW avec insertions incrémentales (hnswlib, Qdrant)
- **1K-100K updates/jour** : Batch processing quotidien/hebdomadaire, blue-green deployment
- **>100K updates/jour** : LSH ou architecture streaming (Kafka + recalcul périodique)

Pattern Delta + Merge : Maintenir un index principal (stable) + index delta (mises à jour récentes). Recherche dans les deux, merge périodique. Utilisé par Elasticsearch, Vespa.

Alternative : Pour datasets très dynamiques, considérer une base vectorielle native cloud (Pinecone, Qdrant Cloud) qui gère automatiquement ces complexités.

> **Ressources open source associées :**
>
> • awesome-cybersecurity-tools — Liste de 100+ outils de cybersécurité

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr