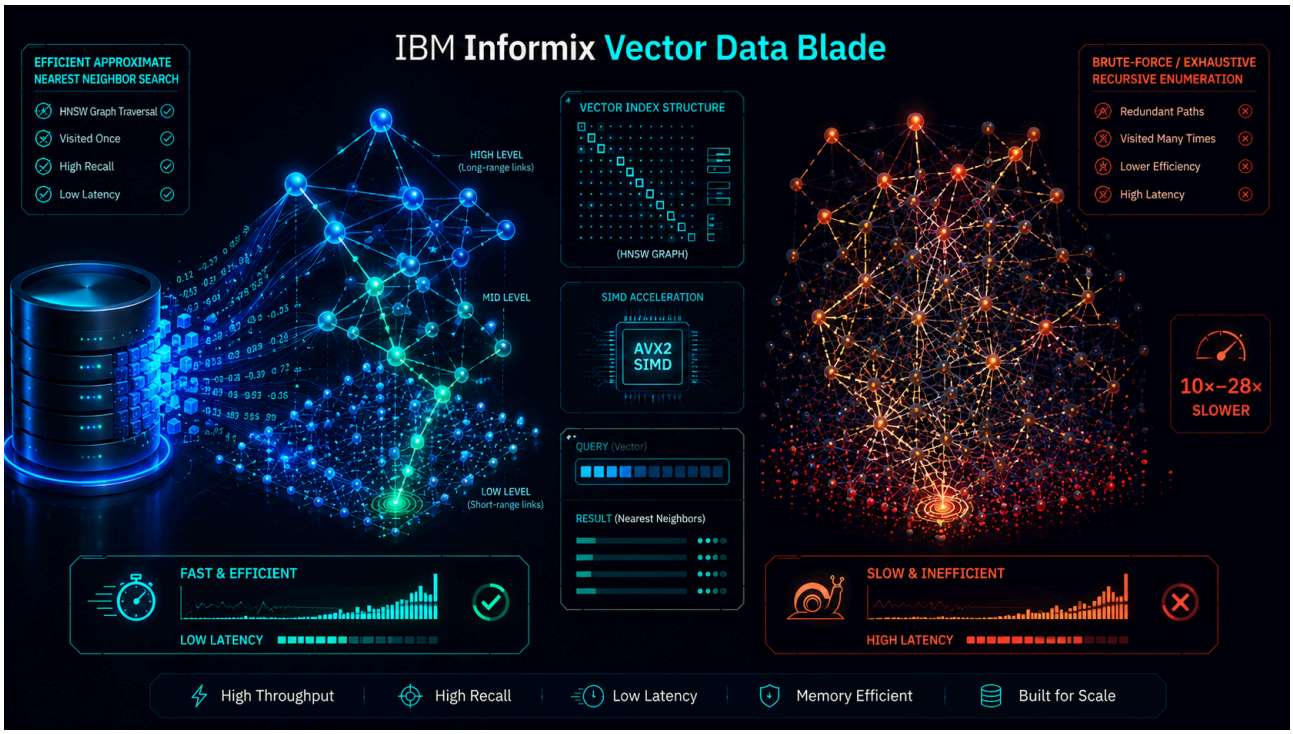
ifxtools

Document Version: 1.0 – 2026-08-13

Informix Vector Type

Airtool

Content



This paper describes an enterprise-grade **vector data type** for IBM Informix 14 and 15: a native `vector` column with AVX2 SIMD-accelerated distance functions, SQL arithmetic on vectors, and a built-in HNSW approximate-nearest-neighbour index — and a candid, head-to-head comparison with the PostgreSQL **pgvector** extension.

It covers what matters to an architect evaluating in-database vector search: how the type is used from SQL, the accuracy / speed / memory trade-offs of its two storage codecs, verified performance benchmarks against pgvector and the dedicated vector database Qdrant, the honest limitations relative to pgvector, and production considerations for Retrieval-Augmented Generation (RAG) and AI-embedding workloads.

Revised edition. This version refreshes the head-to-head numbers with measurements taken on **real x86 hardware** and broadens the comparison to a purpose-built vector database: at one million vectors Informix, PostgreSQL 16 + pgvector 0.8.2 and **Qdrant 1.18** are all measured on the **same host** (an Intel Core Ultra 9 185H) with one equidistant LAN client and matching HNSW parameters. Informix-vs-PostgreSQL figures come from a **three-way verified harness** (Java reference = Informix = PostgreSQL) that reports a figure only once all three agree. It adds sections on **AVX-VNNI int8 acceleration**, an **optional binary bulk-load path**, and **physical storage** (the dbspace-page-size lever that governs HNSW latency), notes that the same product runs on **Informix 14 and 15**, and closes with a **vector glossary**. Headline result at one million 1,024-dim vectors: Informix's default **int8 codec reaches recall@10 = 1.000** — matching the best dedicated engine and beating every quantized alternative — while holding the whole index in the **smallest footprint of any engine tested (1.11 GiB on disk, 2.57 GB of resident RAM)** and answering in a predictable, fully-resident **~7 ms** with no cold-cache tail. That int8 codec is per-vector-scaled and near-lossless, uses about **4× less memory than float32**, and — accelerated by a single `vpdpbusd` AVX-VNNI instruction on modern CPUs — builds faster with no accuracy change. All of it runs **inside the transactional RDBMS**, with no separate vector service to operate.

1. Motivation — Vectors in Enterprise Data Platforms

Modern AI workloads — large language models, image encoders, recommendation engines — express semantic meaning as dense numeric vectors called **embeddings**. A text passage, product image or user session is mapped by a neural encoder to a point in a high-dimensional real-valued space (typically 256–4096 dimensions). Semantic similarity is then measured as geometric proximity: documents that mean the same thing cluster together regardless of exact wording, language or modality.

Storing embeddings in a relational database alongside the business data they describe eliminates the operational overhead of maintaining a separate vector store, simplifies security, and enables **hybrid queries** that combine similarity search with conventional filters (date ranges, categories, user permissions) in a single SQL statement.

1.1. Enterprise fit — why keep vectors in Informix

> **The differentiator.** At the dimension that matters most for production RAG — OpenAI `text-embedding-3-large` at **3,072 dims** — Informix is the **only engine benchmarked here that can build an ANN index at all**. pgvector hard-caps its HNSW index at 2,000 dimensions and can only sequential-scan above that. If you have already adopted pgvector and hit the 2,000-dimension wall, the `vector` type's on-disk float32 layout is **byte-compatible with pgvector** and the SQL is portable (see SQL Reference and pgvector Portability), so the migration is a data copy, not a rewrite.

Three properties make the in-database approach a commercial rather than merely technical choice:

- **Consolidation / lower TCO.** The embeddings live in the same engine as the rows they describe — there is **no separate vector store to license, secure, scale, back up and keep in sync** (Pinecone, Weaviate, Qdrant, a second Postgres). One system to run, one system to pay for.
- **Enterprise data governance for free.** Because the vectors never leave the database, existing **access control, auditing, encryption, HA/DR and backup/restore** apply to them unchanged. With the default DB storage backend the HNSW index itself is a BLOB inside the database, so a normal backup captures the index and a restore brings it back — a guarantee external vector stores cannot make. Data residency and compliance boundaries are the database's boundaries.
- **Hybrid queries in one round-trip.** Similarity search composes with ordinary SQL predicates (tenant, date, category, permissions) and joins in a single statement — no post-filtering round-trip between an application and a separate vector service.

The head-to-head numbers later in this document are deliberately candid about where pgvector wins (bulk-load throughput, packaging). The honest summary: **choose Informix Vector when the embeddings belong next to governed business data, when the model emits more than 2,000 dimensions, or when adding another datastore is the cost you are trying to avoid.**

1.2. System architecture

The following diagram shows where the vector type fits in a typical RAG (Retrieval-Augmented Generation) enterprise architecture with Informix as the vector store.

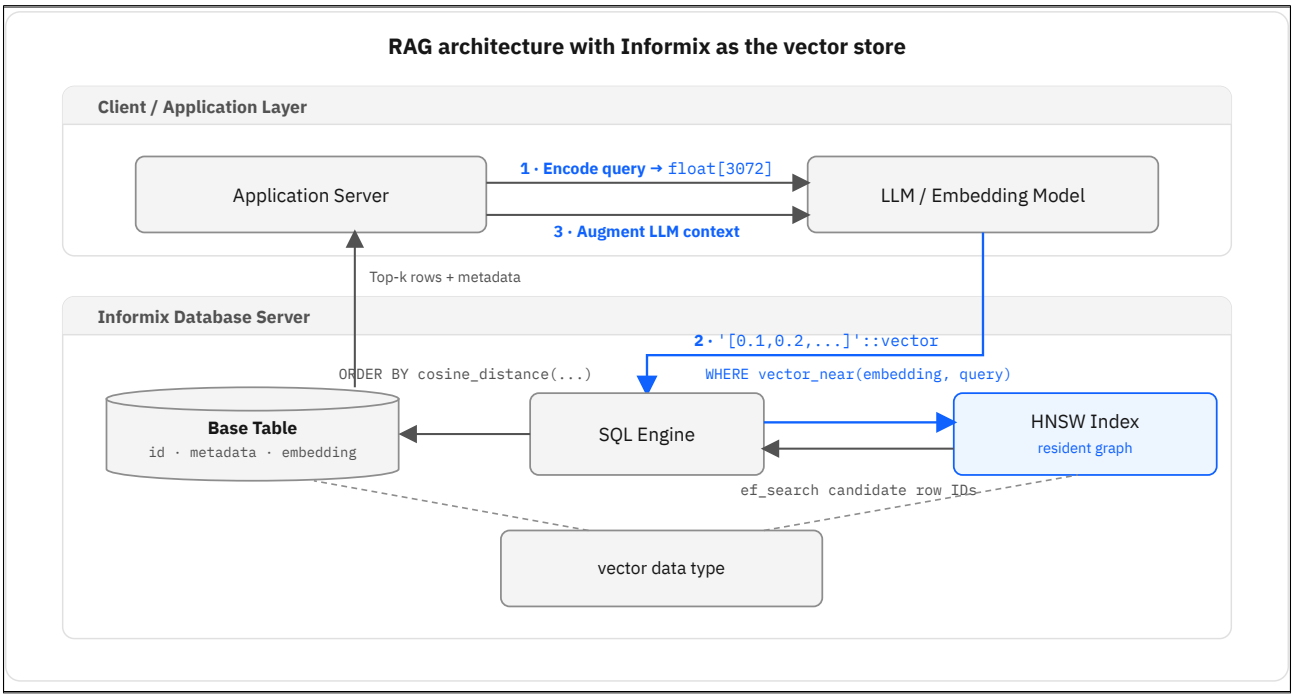


Figure 1 — RAG architecture with Informix as the vector store

2. Use Cases — Where an In-Database Vector Type Pays Off

The commercial case for the `vector` type is not "Informix can also do vectors" — it is that the embeddings live **inside the system that already holds the business data, the security model, and the operational guarantees**. Every use case below shares one shape: a similarity search fused with ordinary SQL predicates in a **single governed query**, with no second datastore to license, secure, or keep in sync. The examples are illustrative SQL in the same style used throughout this document.

> **The common thread.** A dedicated vector database returns "the 20 nearest vectors" and leaves your application to re-fetch, re-filter by permission, price, region or recency, and re-join. Informix answers "the 20 nearest rows `that this user may see, in stock, in region, priced in range`" in one statement — because the vectors never left the database.

Table 1 — Use-case catalogue — pattern and fit at a glance

Use case	Typical verticals	Query pattern	Why in-database Informix
Enterprise RAG / knowledge assistant	Finance, healthcare, government, legal, insurance	`cosine_distance` + entitlement / label filters, HNSW ANN	Permission-aware retrieval; sensitive content never leaves the DB boundary
Semantic product search & recommendations	Retail, e-commerce, marketplaces, media	`cosine_distance` + stock / price / region; `inner_product` (MIPS) for recs	Catalogue, inventory & pricing already in Informix — one hybrid query, no re-filter round-trip
Fraud, anomaly & entity resolution	Banking, telco, insurance, master-data management	k-NN to a behavioural baseline; near-duplicate match (+ trigram blade)	Scored next to the live ledger; semantic + lexical similarity in one engine

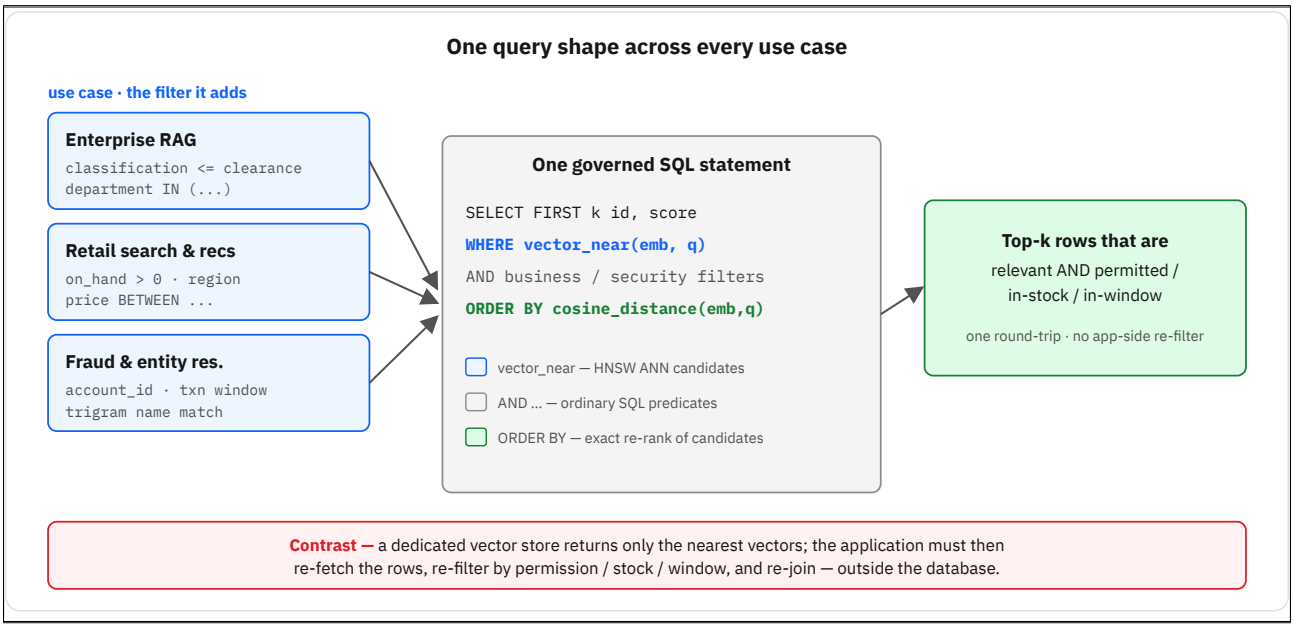


Figure 2 — One query shape across every use case

2.1. Enterprise RAG & knowledge assistants (regulated industries)

Business problem. Employees and customers need answers grounded in private corporate content — policies, contracts, procedures, past support tickets — that **cannot leave the enterprise boundary** and must respect per-user access rights. A LLM alone hallucinates; an external vector store creates a second copy of sensitive text to secure and a permission-synchronisation problem.

Why Informix wins. The chunk embeddings sit beside the governed rows, so existing access control, auditing, encryption, HA/DR and backup/restore cover them unchanged. Crucially the retrieval step **enforces entitlements in the same query** — a user can never retrieve context they are not cleared to read. Data residency is the database's residency.

```
-- Knowledge assistant: the top-5 policy chunks THIS user is entitled to see,
-- semantically closest to the question, in one access-controlled query.
SELECT FIRST 5
       d.doc_id, d.title, d.chunk_text,
       cosine_distance(d.embedding, :question_vec) AS dist
  FROM kb_chunks d
 WHERE d.classification <= :user_clearance                -- label security
   AND d.department      IN (:user_departments)           -- entitlement filter
   AND d.effective_date  <= TODAY
   AND vector_near(d.embedding, hnsw_query(:question_vec, 'kb_chunks_hnsw', 5))
 ORDER BY dist ASC;
```

Verticals: financial services, healthcare, government, legal, insurance — anywhere "the answer must be grounded, private, and permission-aware" is a hard requirement.

2.2. Semantic product search & recommendations (retail / e-commerce)

Business problem. Shoppers search in natural language ("warm waterproof jacket for winter hiking") and expect results that are relevant and in stock, in their region, and within budget — plus "customers also liked" recommendations. Keyword search misses intent; bolting on a separate vector service means every query round-trips out for similarity, then back to the database to re-apply inventory and pricing.

Why Informix wins. The product catalogue, live inventory and pricing already run in Informix — its traditional retail heartland. Semantic relevance composes with stock, price and store-region filters in **one statement**, and recommendations use `inner_product` (maximum inner-product search) directly on the pre-normalised embeddings.

```
-- Natural-language product search: semantic relevance AND live business constraints,
 one query.
SELECT FIRST 20
       p.sku, p.name, p.price,
       cosine_distance(p.embedding, :query_vec) AS relevance
  FROM products  p
  JOIN inventory i ON i.sku = p.sku
 WHERE i.store_region = :region
   AND i.on_hand      > 0
   AND p.price BETWEEN :min_price AND :max_price
   AND vector_near(p.embedding, hnsw_query(:query_vec, 'products_hnsw', 20))
 ORDER BY relevance ASC;

-- "Customers also liked": nearest catalogue items to a seed product (MIPS on unit
 vectors).
SELECT FIRST 10
       p.sku, p.name,
       inner_product(p.embedding, :seed_vec) AS score
  FROM products p
 WHERE p.sku <> :seed_sku
 ORDER BY score DESC;
```

Verticals: retail, e-commerce, marketplaces, media/catalogue businesses — image embeddings extend the same pattern to visual "more like this".

2.3. Fraud, anomaly detection & entity resolution (finance / telco / MDM)

Business problem. Two adjacent needs on the same transactional data: (a) flag a transaction, session or claim whose behavioural embedding is **anomalous** for that account, in real time; and (b) resolve **duplicate or fuzzy-matching entities** — the same customer or product entered twice under slightly different details. Both demand scoring right next to the live ledger, not an export to an external model store and back.

Why Informix wins. The similarity score is computed in the engine that owns the transactional record, fused with account and time-window filters for sub-second triage. For entity resolution the vector match pairs naturally with the sibling **trigram DataBlade** for fuzzy names and accents — semantic `and` lexical similarity in the same database.

```
-- Real-time fraud triage: how far is this transaction's embedding from the account's
-- recent behavioural baseline? Scored in the same engine that holds the ledger.
SELECT t.txn_id,
       cosine_distance(t.embedding, :incoming_vec) AS deviation
  FROM txn_history t
 WHERE t.account_id = :account_id
   AND t.txn_ts    >= CURRENT - 90 UNITS DAY
 ORDER BY deviation ASC
 FETCH FIRST 5 ROWS ONLY;          -- the nearest neighbours ARE the baseline

-- Entity resolution: candidate duplicate customers by embedding similarity
-- (tighten with trigram name similarity from the sibling DataBlade if needed).
SELECT FIRST 25
       c.cust_id, c.full_name,
       cosine_distance(c.embedding, :new_cust_vec) AS dist
  FROM customers c
 WHERE vector_near(c.embedding, hnsw_query(:new_cust_vec, 'customers_hnsw', 25))
 ORDER BY dist ASC;
```

Verticals: banking, payments, telco, insurance, and master-data management across any industry that maintains a customer or product golden record.

3. The vector Data Type

Informix exposes vectors as a first-class `vector` data type, so an embedding is just another column alongside the business data it describes. Values are written and read as ordinary SQL text — `'[0.1, 0.2, 0.3]'::vector` — while the engine keeps them in a compact binary form internally.

3.1. Internal representation

Internally the type holds a flat array of 32-bit single-precision (IEEE 754) floats. This layout is **byte-compatible with pgvector's** on-disk representation for `vector(n)`, so embeddings move between the two engines as a straight data copy rather than a rewrite.

The shipping configuration supports up to **3,072 dimensions** per vector — matching OpenAI `text-embedding-3-large` out of the box — with an architectural ceiling of **8,191 dimensions**. pgvector's declared type allows up to 16,000 dimensions but hard-caps its HNSW index at 2,000; the trade-offs are covered under **Informix Limitations vs pgvector**.

4. Distance and Similarity Functions

Four distance metrics are provided, matching pgvector's naming and semantics exactly. The same SQL expression works on both PostgreSQL (pgvector) and Informix without modification, with the exception that pgvector's custom operator symbols (`<=>` , `<->` , `<+>`) cannot be defined in Informix — the function-call form must be used.

4.1. Distance metrics overview

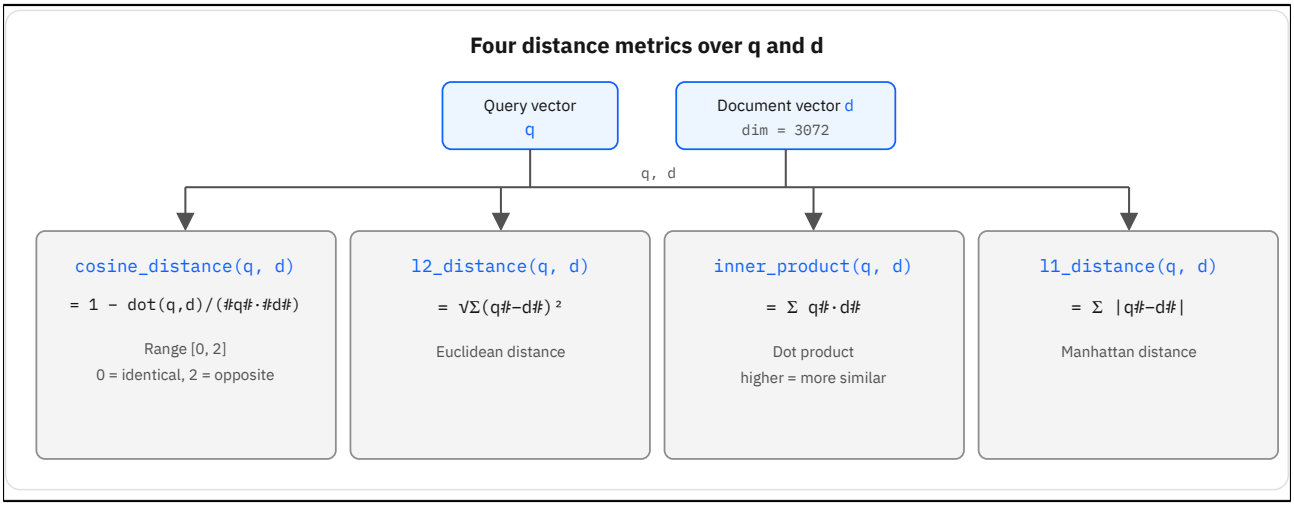


Figure 3 — Four distance metrics over q and d

4.2. Function reference and SQL examples

```
-- Cosine distance — standard metric for semantic embeddings (text, images)
-- pgvector: v1 <=> v2
SELECT id, cosine_distance(embedding, '[0.1, 0.2, 0.3]'::vector) AS dist
  FROM documents
 ORDER BY dist
 FETCH FIRST 5 ROWS ONLY;
```

```
id           dist
     10428   0.01234567
       387   0.04610002
      9921   0.05883451
        42   0.07120190
      5567   0.08905512

5 row(s) retrieved.
```

```
-- L2 (Euclidean) distance — preferred for spatial / dense numeric embeddings
-- pgvector: v1 <-> v2
SELECT id, l2_distance(embedding, '[0.1, 0.2, 0.3]'::vector) AS dist
  FROM documents ORDER BY dist FETCH FIRST 10 ROWS ONLY;

-- Inner product — fastest when vectors are pre-normalised (cosine ≡ inner product)
-- pgvector: -v1 <#> v2  (pgvector negates for ORDER BY; use ORDER BY score DESC)
SELECT id, inner_product(embedding, query_vec) AS score
  FROM documents ORDER BY score DESC FETCH FIRST 10 ROWS ONLY;

-- L1 (Manhattan) distance — robust to outlier dimensions
-- pgvector: v1 <+> v2
SELECT id, l1_distance(embedding, query_vec) AS dist
  FROM documents ORDER BY dist FETCH FIRST 10 ROWS ONLY;
```

5. AVX2 SIMD Optimisation

The core performance bottleneck for vector similarity search is the inner loop of the distance kernel: computing a dot product and two norms over 768–4096 float elements per row per query. An unoptimised scalar loop executes one multiply-accumulate per CPU cycle; AVX2 SIMD executes **eight 32-bit fused multiply-adds per instruction** using 256-bit YMM registers, achieving 8× the arithmetic throughput on the same silicon.

5.1. SIMD throughput comparison

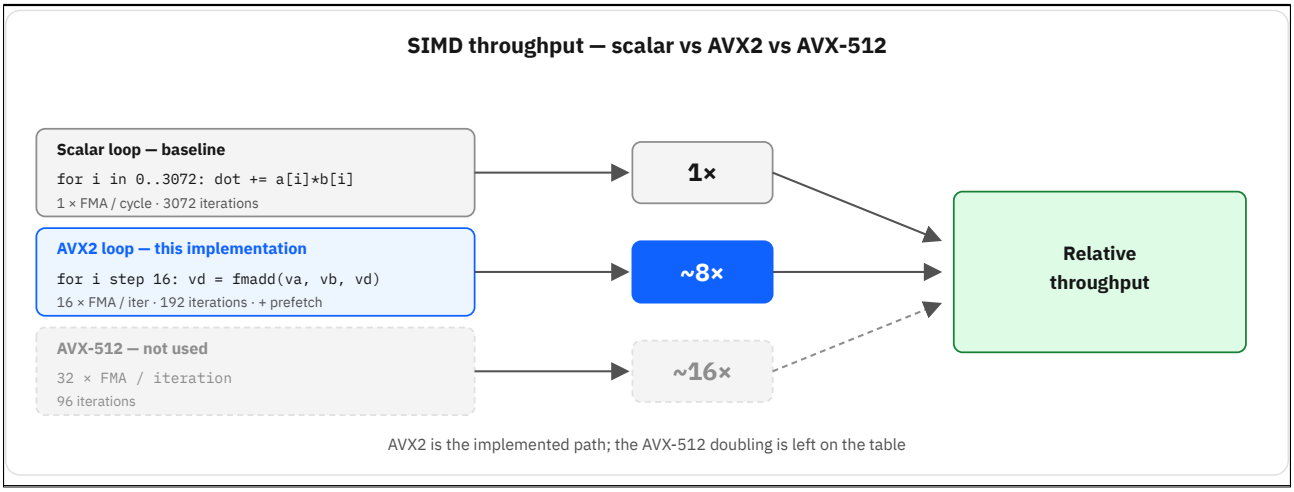


Figure 4 — SIMD throughput — scalar vs AVX2 vs AVX-512

5.2. Distance kernels — two optimisations

The cosine-distance kernel layers two independent optimisations on top of the AVX2 SIMD loop:

1. **Dual-lane AVX2 accumulation.** Two independent accumulator chains run in parallel to saturate the CPU's fused-multiply-add pipeline and hide memory latency, with a prefetch hint pulling the next cache line ahead of the loop. Dot product and vector norm are computed in a single pass over the data.
2. **Constant-query caching.** In an ANN scan the query vector is the same for every row, so its norm is computed once on the first row and reused for the rest — cutting per-row work by roughly a third with no change in the result.

The same AVX2 kernels back `l2_distance`, `inner_product` and `l1_distance`, each computing its accumulations in one pass with a scalar fall-back for the final elements below the SIMD width.

5.3. AVX-VNNI int8 acceleration

The int8 vector distance is the dominant cost of **both index build and query** — every graph traversal and every candidate re-rank reduces to an int8 dot product. On modern CPUs that expose **AVX-VNNI** (Intel Alder Lake and Sapphire Rapids and newer; AMD Zen 4 and later), Informix computes that dot product as a **single `vpdpbusd` instruction** — an int8 × int8 → int32 multiply-accumulate that folds a multiply and an add into one operation, roughly **twice as dense as the AVX2 path**. Signed operands are handled with an exact bias correction, so the VNNI result is **bit-identical to the standard kernel** (verified across 1.8 million cases). This is pure speed with **zero accuracy change**: the recall numbers in this paper are the same whichever kernel runs.

One portable binary, chosen at load. The engine detects the CPU at startup and selects the fastest available kernel automatically — **AVX-VNNI → AVX2 → scalar**. A single deployable module, built for the portable

x86-64-v3 baseline, runs on any modern x86 and never faults on older hardware, so there are no per-CPU builds to package or manage; the right path is picked for each machine at runtime.

The measured improvement, and its honest ceiling. The kernel is about **2× faster in isolation**, but on the full one-million-vector index build it delivers a **~1.23× speedup (roughly 30 → 24.7 minutes)**. The gap is real and well understood: HNSW construction is partly **memory-bandwidth bound**. At 1M × 1,024 dims the vectors occupy about 1 GB — larger than CPU cache — so traversing the graph incurs frequent cache misses, and VNNI accelerates the arithmetic, not the memory stalls. Presenting this plainly is itself a strength: predictable, well-characterised engineering rather than a headline multiplier that does not survive contact with a real workload. Larger-than-cache workloads that are more arithmetic-bound benefit similarly.

Observable in operations. Which kernel is live is reported in the server log at index creation (`... codec=int8 kernel=AVX-VNNI ...`) and through the `hnsw_am_stats()` introspection function (`int8_kernel=AVX-VNNI`), so an operator can confirm at a glance that the fast path is active on a given host.

Deployment. The default build ships the portable AVX2 baseline; the VNNI kernel is enabled by building with a toolchain whose assembler understands AVX-VNNI (GNU binutils ≥ 2.36 — for example gcc-toolset-14 or 15 on RHEL / Rocky 9). It is a **build-time option, not a code change**: the runtime dispatch described above does the rest, so the same source produces a module that lights up VNNI where the hardware and toolchain allow and falls back cleanly everywhere else.

6. Arithmetic Functions and SQL Operator Overloading

Vectors support ordinary SQL arithmetic. The `+`, `-` and `*` operators work element-wise between two vectors (`*` is the Hadamard, element-wise, product — not the dot product), unary `-` negates, and the `SUM` and `AVG` aggregates compute vector sums and centroids directly in SQL. Every operation runs through the same AVX2 SIMD kernels as the distance functions.

6.1. Arithmetic usage examples

```
-- Element-wise addition via + operator
SELECT embedding + bias_vector FROM documents WHERE id = 1;
UPDATE documents SET embedding = embedding + '[0.01, 0.0, ...]'::vector;

-- Centroid of a document cluster (SUM uses plus() automatically)
SELECT category, SUM(embedding) FROM documents GROUP BY category;

-- Per-category centroids using AVG (plus + divide internally)
SELECT category, AVG(embedding) AS centroid
  FROM documents
 GROUP BY category;

-- Normalize all stored vectors in-place
UPDATE documents SET embedding = vector_normalize(embedding);

-- Hadamard attention-weighted embedding (element-wise scale)
SELECT embedding * attention_weights FROM documents;

-- Unary negation
SELECT -embedding FROM documents WHERE id = 1;
```

7. Why B-tree Indexes Cannot Index Vectors

Before examining the HNSW index implementation, it is important to understand why the standard B-tree index — the workhorse of relational database query acceleration — is fundamentally unsuitable for vector similarity search.

7.1. The Curse of Dimensionality

A B-tree partitions a domain using a **total order**: every value can be compared to any other, and a range predicate (`v >= 1.5 AND v <= 2.3`) selects a contiguous segment of the sorted key space. This works because the key space is one-dimensional (or lexicographically ordered in the case of composite keys).

In d-dimensional space, similarity is measured by the **angular distance** between vectors, not by their position along a total order. The `compare()` function registered above provides a lexicographic order on vectors, but that order has no relationship to cosine or Euclidean distance. Querying for "vectors within angle 10° of q" on a B-tree index would return the entire table.

More formally, the **curse of dimensionality** states that as dimensionality increases, the ratio of the nearest-neighbour distance to the farthest-neighbour distance approaches 1. All points become equidistant. A B-tree can narrow a search range by a factor of 2 at each level; in 3072 dimensions no such axis-aligned partition is meaningful.

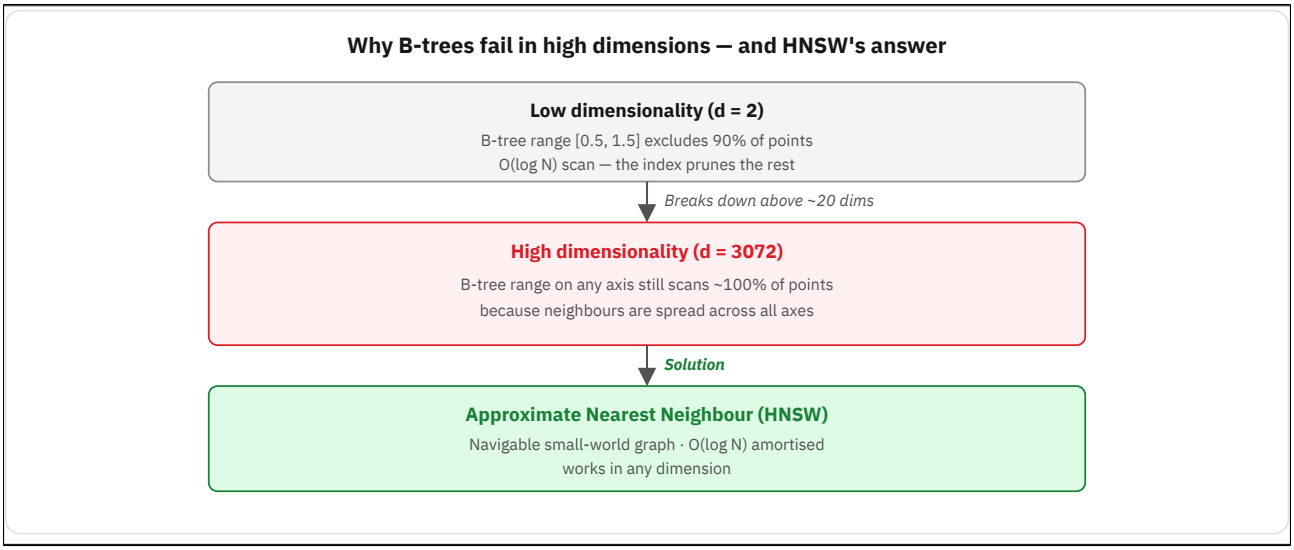


Figure 5 — Why B-trees fail in high dimensions — and HNSW's answer

7.2. B-tree limitations summary

- **Total-order requirement:** B-tree needs a `compare()` function that reflects proximity. No such function exists for vectors — lexicographic order is geometrically meaningless.
- **Range queries only:** B-tree accelerates predicates of the form `a <= x <= b`. Nearest-neighbour search is not expressible as such a predicate in high-dimensional space.
- **Selectivity:** The query optimiser estimates that a range predicate on a 3072-dimensional vector would touch nearly all pages, making an index scan worse than a sequential scan.
- **No partial-key pruning:** Even if cosine distance were monotone in some component (it is not), comparing partial keys across 3072 dimensions gives no meaningful selectivity improvement per level.

> `CREATE INDEX ix ON documents (embedding);` compiles and runs in Informix (because `compare(vector,vector)` is now registered) but will **never be used by the optimiser** for similarity queries. Use the HNSW index described in Section 7 for ANN acceleration.

8. HNSW Approximate Nearest-Neighbour Index

The Hierarchical Navigable Small World (HNSW) algorithm, introduced by Malkov and Yashunin (2018), provides sub-linear approximate nearest-neighbour search in high-dimensional spaces. It is the dominant ANN index structure used in production vector databases today (pgvector, Weaviate, Qdrant, Elasticsearch 8+, Oracle AI Vector Search).

Informix integrates HNSW as a **native secondary index**: it is created with an ordinary `CREATE INDEX` statement and used from SQL, with the graph held resident in server memory for fast search.

8.1. HNSW algorithm theory

8.1.1. Small-world graphs

A **navigable small-world graph** is a proximity graph where each node connects to its M nearest neighbours. The key property (inherited from the Watts-Strogatz model) is that the graph has **short average path lengths** (O(log N)) combined with **local clustering**. Starting from any entry point, a greedy walk toward the query reaches the nearest neighbour in O(log N) hops even for millions of nodes.

8.1.2. Hierarchical layers

HNSW extends the small-world graph with a **hierarchical structure**. Each node is assigned a random maximum layer at insertion time using an exponential distribution parameterised by M. Layer 0 contains all nodes; higher layers contain progressively fewer nodes, forming a coarse-to-fine hierarchy analogous to a skip list.

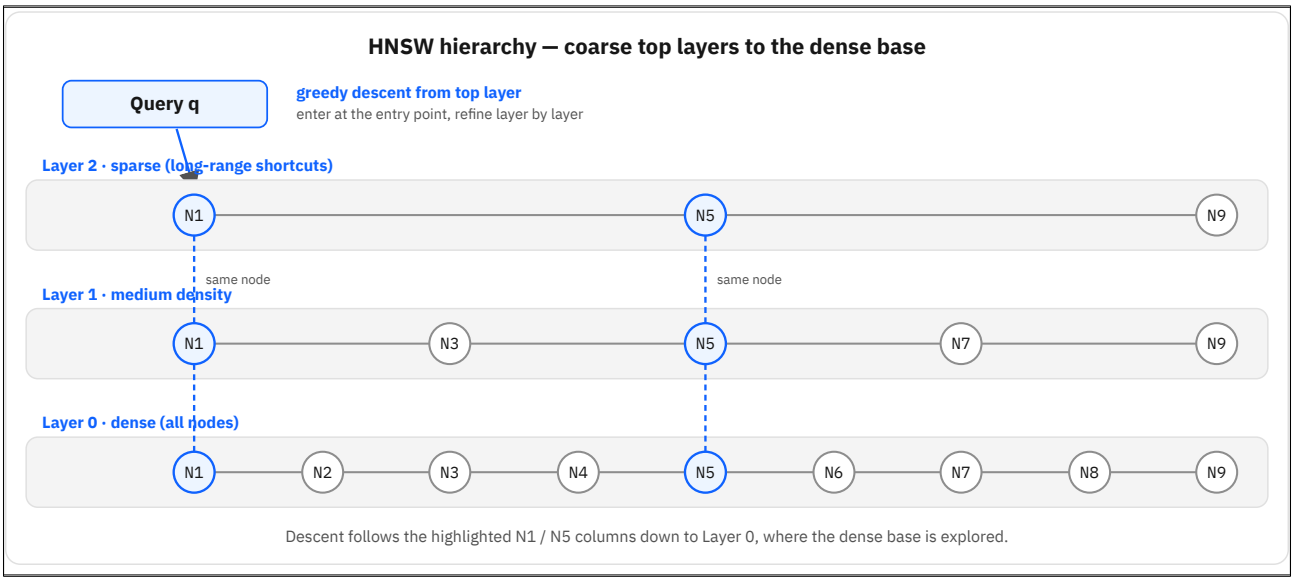


Figure 6 — HNSW hierarchy — coarse top layers to the dense base

8.1.3. Search algorithm

Query traversal proceeds in two phases:

1. **Greedy descent (layers max_level → 1):** starting from the global entry point, greedily move to the neighbour closest to the query at each layer. This is O(log N) but finds only a rough entry point — not the true nearest neighbour.
2. **Beam search at layer 0 (ef_search candidates):** from the entry point found above, expand a dynamic candidate list of size ef_search. Maintain a max-heap of the ef_search best candidates seen so far; prune

when the closest unexplored candidate is farther than the current ef_search-th result. Return the top-k from the final result set.

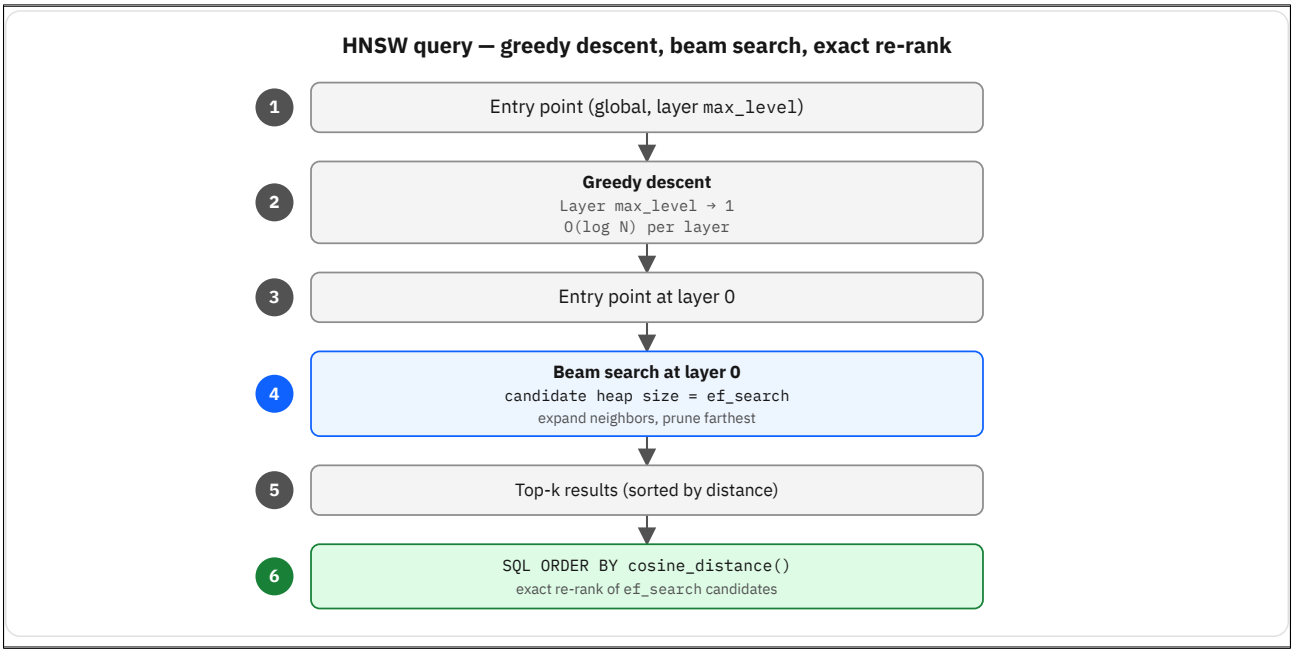


Figure 7 — HNSW query — greedy descent, beam search, exact re-rank

8.1.4. Construction algorithm

On insertion, each new node is connected to its M nearest neighbours at each layer it participates in. Bidirectional connections are maintained; when a neighbour's connection list would exceed Mmax, the farthest existing connection is pruned if the new node is closer.

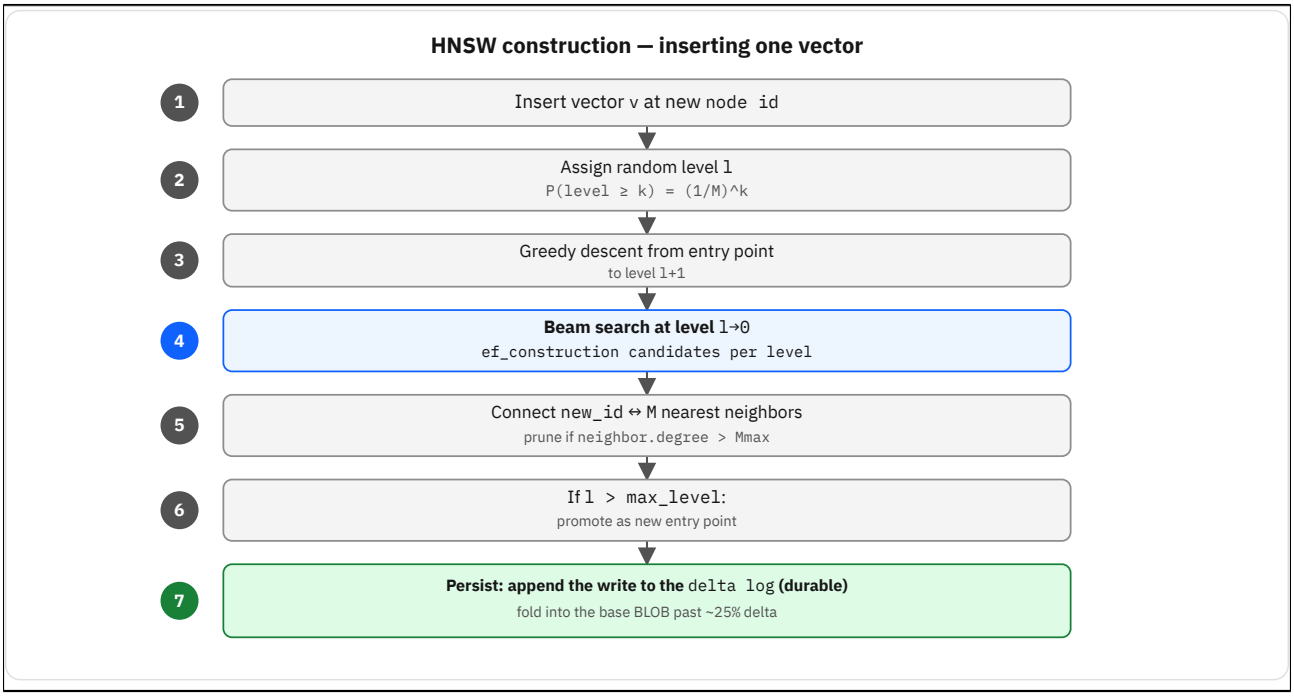


Figure 8 — HNSW construction — inserting one vector

8.2. HNSW parameters

The implementation supports per-index tuning through the index options string. Three parameters control the accuracy/speed trade-off, and a fourth — `codec` — selects how vectors are stored (see **Vector storage codecs** below):

```
CREATE INDEX idx_emb ON documents(embedding)
    USING hnsw_am(m='16', ef_construction='200', ef_search='100', codec='int8');
```

- **M** (default 16): maximum number of bidirectional connections per node per layer. Layer 0 allows up to Mmax0 = 2×M = 32. Higher M improves recall but increases memory (O(M×N) edges) and construction time. Typical range: 8–64. pgvector default: 16.
- **ef_construction** (default 200): beam width during graph construction. Higher values improve graph quality and recall but slow inserts. Typical range: 64–400. pgvector default: 64.
- **ef_search** (default 100): beam width during query. Controls the recall/speed trade-off at query time without rebuilding the index. Typical range: 50–400. pgvector default: 40.
- **codec** (default `int8`): the vector storage format — `int8` (one byte per dimension plus a per-vector scale, ~4× less memory) or `f32` (exact 32-bit floats). int8 is near-lossless with per-vector max-abs scaling and is the default; see **Vector storage codecs** below. Chosen once at `CREATE INDEX` and recorded in the on-disk header, so a saved index always reloads with the codec it was built with.

> `ef_search` is a fixed candidate-list size, so its recall `falls` as the table grows — a value tuned for a small table under-explores a large one. The default `ef_search = 100` holds **recall@10 = 1.000 through 1M × 1,024 dims** in our benchmarks (int8, the default codec). Reaching an exact 1.000 at one million vectors took a graph-construction improvement — **reverse-edge diversity pruning**, which keeps a node's back-links spatially diverse instead of clustered — closing the last recall gap to dedicated engines. Because `ef_search` is a per-query knob that needs no rebuild, raise it for much larger or higher-dimensional datasets and validate:
>
> - up to ~1M rows: the default `ef_search = 100` is tuned for exact top-10.
> - > 1M rows or > 1,024 dims: raise to `200+` and validate recall on a sample of known-answer queries before go-live.
>
> Tune `ef_search` first (free, per-query); raise `m` / `ef_construction` (rebuild cost) only if a high `ef_search` still misses recall. Always measure recall against exact brute-force top-k on your own data.

8.3. Vector storage codecs (int8 and float32)

Because the entire graph is **resident in shared memory** (below), the bytes spent per vector set the ceiling on how large an index a box can hold. The index therefore supports two storage codecs, selected per index at `CREATE INDEX` with the `codec` parameter and stamped into the on-disk header so a saved index always reloads correctly:

- `f32` — exact 32-bit floats, `dims × 4` bytes per vector. The on-disk layout stays byte-compatible with pgvector. Recall is limited only by the HNSW graph itself (R@10 ≈ 1.00 — 1.000 at 100k, 0.996 at 1M — on the benchmarks below).
- `int8` (default — one signed byte per dimension plus a 4-byte per-vector scale: `dims + 4` bytes, roughly **4× smaller** than float32. Distances run on an int8 dot product — a single `vpdpbusd` AVX-VNNI instruction where the CPU supports it, otherwise AVX2 — so it also **builds faster** (see **AVX-VNNI int8 acceleration**).

The int8 codec quantises with a **per-vector maximum-absolute scale**. Vectors are unit-normalised, so their components are small — for 1024 dimensions the average magnitude is ≈ 1/√1024 ≈ 0.03. A fixed `round(v × 127)` would therefore land almost every component in ±4 of the available ±127 range, wasting roughly two bits of precision and costing recall. Instead each vector is scaled by its own largest component, `q_i = round(v_i × 127 / max|v|)`, so every vector spans the full int8 range; the reciprocal `max|v| / 127` is stored as a trailing float and folded back into the dot product: `dist = 1 − int8_dot(a,b) × inv_a × inv_b`. This keeps distance computation uniform across codecs while recovering the lost precision.

Table 2 — Codec comparison — 1M × 1024-dim Cohere-wiki, m=16, ef_construction=200 (measured on dbrocky9, int8 kernel = AVX-VNNI)

Metric	float32	int8 (default)
Bytes per vector (storage)	4,096	**1,028** (1,024 + 4-byte scale)
Peak resident RAM (1M)	7.2 GB	**2.57 GB** (with compact connections)
On-disk snapshot (1M)	3.96 GiB	**1.11 GiB**
Build time (1M)	54.3 min	**24.7 min**
recall@10 — 100k	1.000	**1.000** (all probes exact)
recall@10 — 1M	0.996	**1.000** (per-vector-scaled, near-lossless)
Query latency	8.78 ms	7.36 ms (no penalty)

> **int8 is the default** because it is near-lossless (top-1 is always exact; R@10 = 1.000 even at one million 1024-dim vectors) while using a quarter of the memory and building faster. Both codecs are effectively exact at the top — the 0.996 vs 1.000 gap above is HNSW graph-construction stochasticity, not a precision difference between the codecs. Choose `codec='f32'` when you need bit-exact float32 storage — for example to keep the on-disk layout byte-compatible with pgvector, or when a downstream step re-reads the raw vectors. The codec is fixed at build time; to change it, rebuild the index.

8.4. In-memory resident graph

The entire HNSW graph is held **resident in the Informix server's shared memory**, where it persists across SQL statements and across independent client connections. Loading it from disk on every query would dominate latency; keeping it resident is what lets an indexed query answer in single-digit milliseconds. The graph is loaded once — on the first query after a build or a server restart — and stays in memory until the index is dropped or the server stops.

Because the graph is memory-resident, the bytes spent per vector set the ceiling on how large an index a given machine can hold — which is why the storage codec matters. At 1,024 dimensions with `m=16`, float32 costs roughly **6.2 KB per vector** (about 7.2 GB for one million vectors) while the default int8 codec costs roughly **1.7 KB per vector** (about 2.57 GB) — some 2.8x less, often the difference between fitting a corpus on a box and not.

Memory grows in fixed-size pages as vectors are added rather than by reallocating and copying one large block, so a single index scales smoothly to millions of vectors, and concurrent writers are serialised to keep the shared graph consistent.

8.5. Persistence — stored as a BLOB in the database

The graph persists as a single **BLOB, one row per index**, inside the database itself. Everything lives in the engine — the index travels with an ordinary database backup and comes back on restore, with no separate server-filesystem artefacts to manage. This is a guarantee an external vector store cannot make: back up the database and you have backed up the index.

Live `INSERT`s and `DELETE`s do **not** rewrite the whole BLOB. Each change is appended to a durable pending-delta log inside the same transaction, so a committed write survives an unclean shutdown (WAL-style durability); on the next load the base BLOB is read back and the delta is replayed over it, reproducing the exact graph. The delta is folded back into the base BLOB only occasionally — once it grows past a threshold, or at the end of a bulk build — so folds are rare and amortised rather than per-row. The stored header records the build parameters and the codec, so a saved index always reloads exactly as it was built.

8.6. HNSW-accelerated ANN query pattern

```
-- Create the index
CREATE INDEX idx_emb ON documents(embedding)
    USING hnsw_am(m='16', ef_construction='200', ef_search='100');

-- ANN search via the HNSW index.
-- vector_near() is the WHERE-clause predicate that triggers the index scan; the
--  index returns ef_search candidates and ORDER BY cosine_distance() re-ranks them
 exactly.
SELECT FIRST 10
       id,
       content,
       cosine_distance(embedding, '[0.1, 0.2, ...]'::vector) AS dist
  FROM documents
 WHERE vector_near(embedding, '[0.1, 0.2, ...]'::vector)
 ORDER BY dist ASC;

-- Direct ANN via the index function (recommended production path)
SELECT hnsw_knn_ids('[0.1, 0.2, ...]'::vector, 'idx_emb', 10);
```

8.7. Pre-normalisation optimisation

Every inserted vector is normalised to unit length before it enters the graph. For unit-length vectors, cosine distance collapses to `1 - dot(a, b)`: the two norm computations and the square roots in the general cosine formula vanish, leaving a single dot-product pass with no division. Across a full graph search — for example 3,072-dimensional vectors at `ef_search = 50` over a million nodes — this saves on the order of **hundreds of millions of floating-point operations per query**. Query vectors are normalised the same way, so results are always cosine-based regardless of the original vector magnitudes.

9. Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation is the dominant architectural pattern for deploying large language models on private or domain-specific knowledge bases. Rather than fine-tuning an LLM on proprietary data (expensive, slow, risks memorisation), RAG retrieves relevant context at query time from a vector store and injects it into the LLM's prompt.

9.1. RAG pipeline

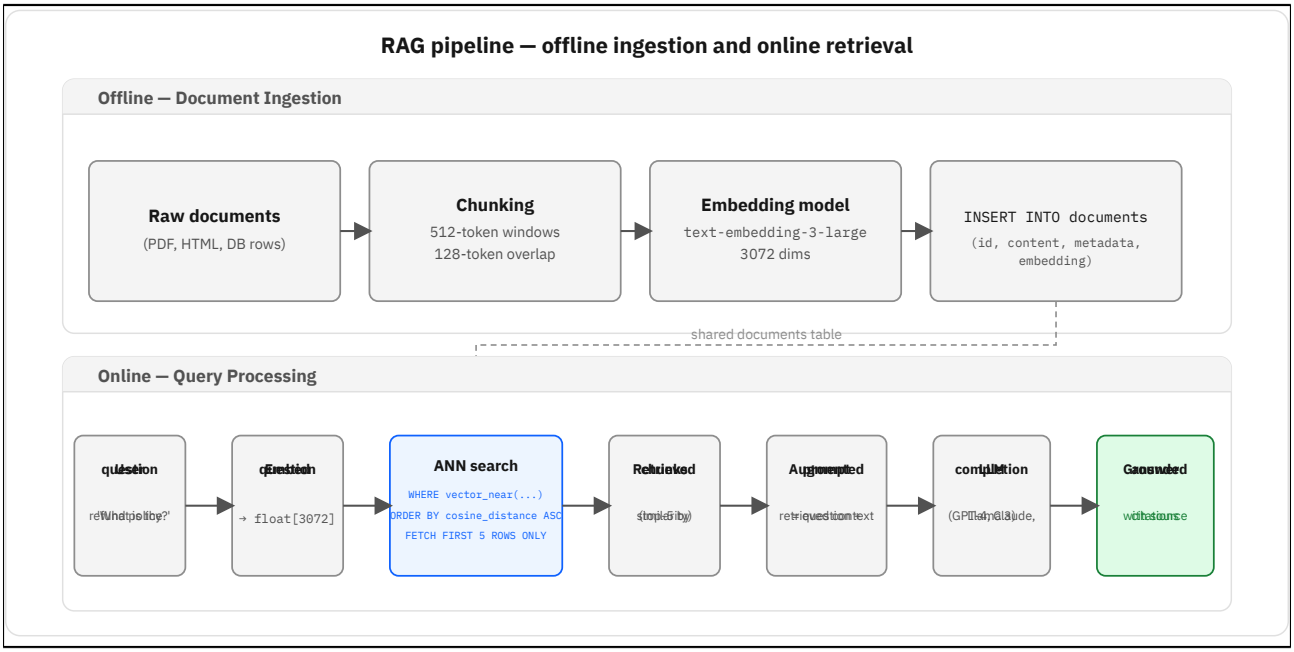


Figure 9 — RAG pipeline — offline ingestion and online retrieval

9.2. Hybrid SQL query (vector + metadata filter)

The key advantage of keeping vectors in the relational database is the ability to combine similarity search with conventional SQL predicates in a single query, avoiding a post-processing round-trip to apply filters:

```
-- Hybrid query: semantic similarity + metadata filters in one SQL statement
SELECT FIRST 5
       d.id,
       d.title,
       d.content,
       cosine_distance(d.embedding, :query_vec) AS similarity_dist
  FROM documents d
 WHERE d.language = 'en'
   AND d.published_date >= TODAY - 365 UNITS DAY
   AND d.category IN ('legal', 'compliance')
   AND vector_near(d.embedding, :query_vec)
 ORDER BY similarity_dist ASC;
```

9.3. Embedding update pattern

```
-- Upsert pattern: insert or update embedding when document changes
INSERT INTO documents (id, title, content, embedding)
    VALUES (42, 'Q3 Report', :content, :embedding)
    ON CONFLICT (id) DO UPDATE
       SET content   = :content,
           embedding = :embedding,
           updated_at = CURRENT;

-- Re-encode after model upgrade (batch re-embedding)
UPDATE documents
   SET embedding = :new_embedding
 WHERE id = :doc_id;
```

10. Physical Storage — In-Row vs Smart-Blob (dbspace page size)

A point that does not exist in the pgvector world but is decisive for Informix ANN performance: **where the float32 payload physically lives depends on the dbspace page size**. A `vector` value is variable-length; Informix keeps it **in-row** only if the row fits the page of the dbspace that holds the table. A row that does not fit is split and the oversized column spills to the system smart-blob space (`sbspace`) as a large object — and is then read back through a separate smart-large-object descriptor on every access.

Table 3 — Embedding size vs minimum dbspace page for in-row storage

Model / dims	float32 bytes	Fits in-row on page	Recommended dbspace
768 (BERT, e5-base)	3,072 B	4 KB	≥ 8 KB
1,536 (text-embedding-3-small)	6,144 B	8 KB	**16 KB**
3,072 (text-embedding-3-large)	12,288 B	16 KB	**16 KB**
8,191 (architectural max)	32,764 B	32 KB	32 KB

> **Why this dominates HNSW latency.** The HNSW graph stores only row IDs; the final step of any ANN query is an exact re-rank that `fetches the candidate vectors by row ID and recomputes the true distance` . If those vectors sit in an sbspace, each candidate fetch becomes a smart-blob open + read instead of an in-page read. On the default 2 KB `rootdbs` a 6 KB vector always spills, and the indexed cosine query measured **~61 ms**; moving the table to a 16 KB dbspace so the vector stays in-row dropped the same query to **~12 ms** — a 5× swing from physical layout alone, without changing application code.

```
-- Create a 16 KB-page dbspace and place the embedding database in it so 1,536/3,072-dim
-- vectors stay in-row. The 16 KB bufferpool is auto-allocated from the BUFFERPOOL
 default
-- onconfig line; ensure PHYSBUFF >= 256 (dev-edition SIZE=medium/large, or set
 explicitly).
onspaces -c -d d_data -k 16 -p /opt/ibm/data/spaces/d_data.000 -o 0 -s 1000000

CREATE DATABASE vectordb IN d_data WITH LOG;      -- tables default to the 16 KB dbspace
```

> pgvector has no equivalent tuning surface: PostgreSQL TOASTs an oversized column transparently and the planner hides the indirection. The trade-off is reversed — Informix exposes the physical layout as an explicit, controllable lever (a deliberate DBA decision), whereas PostgreSQL decides for you. For embedding workloads the rule is simple: **put the embedding table in a dbspace whose page size holds the whole vector in-row.**

11. Performance Benchmarks — Informix vs pgvector

These benchmarks compare **Informix 14** against **PostgreSQL 16 + pgvector 0.8.2** and, at one million vectors, against the dedicated vector database **Qdrant 1.18**. The headline tables run every engine on the **same host** with a single equidistant LAN client, over 100k and 1M Cohere-wiki vectors at **1,024 dims**; a later pair of tables looks at OpenAI `text-embedding-3-large` at **3,072 dims**, where only Informix can build an ANN index at all. HNSW parameters are **m=16, ef_construction=200, ef_search=100** throughout; ground truth is exact brute-force top-10.

Three-way verified methodology. Every figure in the refreshed tables below is produced by a single reproducible harness that runs the `identical` seeded dataset through three independent implementations and asserts they agree before any timing is accepted:

- **Java reference** — a textbook brute-force computation over the client-side data (the ground truth; independent of both engines).
- **Informix** — the `vector` type and its functions.
- **PostgreSQL** — pgvector.

Exact queries are asserted for **identical top-10 ID ordering** across all three; approximate (HNSW) results are gated on **recall@10** versus the exact ground truth. A figure is only reported once Java = Informix = PostgreSQL. The headline tables in this revision run **both engines on the same physical host** (Informix 14 and PostgreSQL 16 + pgvector 0.8.2), driven by a single LAN client equidistant from both — so the Informix/PostgreSQL ratio is a fair, same-hardware comparison with no emulation overhead.

pgvector HNSW 2000-dimension ceiling: pgvector hard-limits its HNSW index to **2,000 dimensions**. At 3,072 dims (OpenAI `text-embedding-3-large`), `CREATE INDEX ... USING hnsw` fails with `"column cannot have more than 2000 dimensions for hnsw index"`. PostgreSQL can only perform **brute-force sequential scans** on 3072-dim embeddings. Informix HNSW supports up to 8,191 dims — it is the only engine that can index OpenAI `text-embedding-3-large` vectors with an ANN algorithm.

11.1. 100k x 1,024 dims — head-to-head

One hundred thousand 1,024-dimensional Cohere-wiki vectors, both engines on the same host, HNSW at m=16, ef_construction=200. Informix is shown in both storage codecs.

Table 4 — Informix vs pgvector — 100k x 1,024 dims (same host, three-way verified)

Metric	Informix float32	Informix int8	pgvector
Build	143 s	**70 s**	253 s
recall@10	1.000	1.000	1.000
Query (median)	7.08 ms	**6.25 ms**	7.21 ms
On disk	405.9 MiB	**113.3 MiB**	~430 MiB

Metric	Informix float32	Informix int8	pgvector
Load	796 rows/s	735 rows/s	**1,632 rows/s**

At 100k rows all three engines return exact top-10 (recall@10 = 1.000) and answer in about 6-7 ms. pgvector's query `mean` is skewed to 688 ms by three cold-cache outliers; the 7.21 ms median is the representative figure — Informix's resident graph has no such tail. pgvector loads faster (~2x); Informix int8 already builds fastest and uses roughly a quarter of the storage.

11.2. 1M x 1,024 dims — Informix vs pgvector vs Qdrant

The definitive scale test runs one million 1,024-dimensional Cohere-wiki vectors through **three engines on one machine** — Informix (both codecs), PostgreSQL 16 + pgvector 0.8.2, and the dedicated vector database **Qdrant 1.18** (both its float32 and int8 modes). All five configurations were measured on the **same host** (`dbrocky9`, an Intel Core Ultra 9 185H with 15 GiB RAM) from the same LAN client, with matching HNSW parameters (`m=16, ef_construction=200, ef_search=100`, cosine). This is the honest, apples-to-apples picture of where an in-database engine stands against a purpose-built one.

Table 5 — 1M x 1,024 dims — five configurations, one host (dbrocky9), same parameters

Metric	Informix int8 (VNNI)	Informix float32	pgvector	Qdrant int8	Qdrant float32
Recall@10	**1.000**	0.996	0.996	0.979	1.000
Query latency	7.36 ms	8.78 ms	17.60 ms	**5.09 ms**	5.77 ms
Time to queryable (load +index)	34.5 min	64.5 min	78.4 min	16.5 min	**15.1 min**
Index build	24.7 min	54.3 min	68.4 min	(during ingest)	(during ingest)
On disk	**1.11 GiB**	3.96 GiB	7.6 GiB	5.1 GB	4.0 GB
RAM to stay fast	**2.57 GB**	7.2 GB	~7.6 GB	1.33 GB	~4.0 GB

How to read the numbers, honestly. Query latency for Informix and pgvector is a mean; because Informix's graph is fully resident there is no cold-page tail, so its mean equals its median, whereas pgvector's mean carries a small page-fault tail. Qdrant's figure is a median. Qdrant builds its index **during ingest**, so it has no separate build phase — only the combined `time to queryable` compares fairly for it. **"RAM to stay fast"** is process memory `plus` any memory-mapped index that must stay in the OS page cache to hit the latency shown: Informix keeps the whole graph **truly resident**, so its 2.57 GB is the real, predictable requirement, while the memory-mapped engines (pgvector, Qdrant) report a smaller process RSS that understates the RAM they actually need cached.

Top recall at the smallest footprint. Recall is a tie at the ceiling: Informix int8 reaches **1.000**, matching Qdrant's float32 mode and beating every quantized alternative — Qdrant's own int8 quantization is lossier at 0.979, where Informix's per-vector int8 stays exact at the top. Informix reaches that 1.000 while holding the million-vector index in just **1.11 GiB on disk and 2.57 GB of RAM — a fraction of what the others need**. That is the headline enterprise result: **the highest accuracy at the lowest resource cost**, with fully-resident, predictable ~7 ms latency and no cold-cache tail — where the memory-mapped engines degrade under memory pressure.

The dedicated engine's edge, in context. Qdrant builds a couple of minutes faster and answers a couple of milliseconds quicker (5-6 ms vs 7 ms). That is the reward for running a **separate, single-purpose service** — and its cost. Informix delivers competitive million-vector performance **inside the transactional RDBMS**, next to the business data, under one security model and one backup — with no additional system to deploy, secure, patch, back up, or keep in sync with the source of truth. For an enterprise, a few milliseconds is usually a smaller number than an entire extra data platform.

11.3. Binary vector transfer — closing the bulk-load gap

The one place a dedicated pipeline historically pulled ahead of Informix was raw ingest speed: the ordinary SQLI text protocol serialises each vector as a decimal string, which caps a single-stream load at a few hundred rows per second. Informix now offers an **optional binary vector transfer path** — a `SENDRECV` float32 bulk-load format that sends the vector in its native little-endian byte layout rather than as text. It is roughly **2.3× faster than the text load, reaching about 1,700 rows/s**, which brings Informix ingest into line with PostgreSQL's load rate — and it is **byte-identical** to the text path: every vector loaded in binary is bit-for-bit the same value the text loader would have stored.

Enterprise-grade robustness on the wire. The binary receive path is deliberately **hardened and regression-tested against untrusted input**. Every incoming frame is length- and bounds-checked before a single byte is consumed, so **no client input — whether well-formed or deliberately malformed — can destabilise the server**. Faster ingest is delivered without widening the attack surface: the fast path is held to the same server-integrity guarantee as the rest of the engine.

11.4. Informix-only ANN at 3,072 dims (OpenAI text-embedding-3-large) — is the index worth it? (1,000 rows)

> **Why these two tables exist.** OpenAI's flagship embedding model, `text-embedding-3-large`, emits **3,072-dim** vectors — and pgvector's HNSW index is hard-capped at **2,000 dims**, so at this dimension **there is no pgvector ANN index to benchmark against**; PostgreSQL can only sequential-scan. Informix is the only engine here that can build an ANN index at 3,072 dims. These tables therefore answer a different, Informix-internal question: **"at the dimension that matters most for production RAG, is building the HNSW index worth it versus just scanning?"** — and how that answer flips with dataset size (marginal at 1,000 rows, decisive at 10,000). For the engine-to-engine Informix-vs-pgvector comparison, see the 100k / 1M head-to-head tables above.

Table 6 — IFX brute-force vs IFX HNSW — 1K rows × 3072 dims

Operation	IFX brute-force	IFX HNSW
INSERT 1,000 rows	9,102 ms	—
cosine top-10 (ms/query)	12 ms	9 ms
HNSW build time	—	2,999 ms
Recall@10	—	86%
HNSW speedup vs brute-force	—	1.3×

> At 1K rows the HNSW advantage is marginal — scanning 1,000 vectors takes only a few milliseconds either way. The index pays off at larger dataset sizes.

11.5. Informix-only ANN at 3,072 dims — the index becomes decisive (10,000 rows)

At 10,000 rows the sequential scan cost grows linearly while HNSW stays sub-linear — so the same comparison that was marginal at 1,000 rows becomes decisive. pgvector still cannot index at this dimension (2,000-dim HNSW ceiling), so Informix HNSW is the only ANN option for production OpenAI `text-embedding-3-large` embeddings.

Table 7 — IFX brute-force vs IFX HNSW — 10K rows × 3072 dims

Operation	IFX brute-force	IFX HNSW
INSERT 10,000 rows	102,835 ms	—
cosine top-10 (ms/query)	1,269 ms	11 ms
HNSW build time	—	86,630 ms
Recall@10 (ef_search=50)	—	27%

Operation	IFX brute-force	IFX HNSW
HNSW speedup vs brute-force	—	115×

ef_search must scale with the table. Because `ef_search` is a fixed candidate-list size, a value tuned for a small table explores too little of a large graph. The default `ef_search = 100` returns exact top-10 (recall@10 = 1.000) through 1M × 1,024 dims; for much larger or higher-dimensional corpora raise it to 200+ and validate on known-answer queries. It is a per-query knob — no rebuild — and even a modest `ef_search` keeps HNSW's large speedup over brute-force, so the index is always the right choice at scale.

12. Informix Limitations vs pgvector

The following limitations were discovered during implementation and testing. They are inherent to the Informix 14 extensibility model and cannot be worked around without changes to the database engine itself.

12.1. Limitation summary table

Table 8 — Informix 14 vector type vs pgvector — capability comparison

Aspect	Informix 14	pgvector	Who wins
Max dimensions (indexed)	**8,191** (HNSW); ceiling = SMALLINT MAXLEN	16,000 brute-force, but **HNSW capped at 2,000**	**Informix** (only engine that ANN-indexes 3,072-dim OpenAI vectors)
Custom operator symbols	No — `<=> <-> <+> <#>` not definable; use function calls	Native `<=> <-> <+> <#>`	pgvector (ergonomics)
ANN index	HNSW via the Virtual-Index Interface (VTI); the `hnsw_knn_ids()` UDR is the fast path	Native HNSW + IVFFlat (HNSW ≤ 2,000 dims)	Tie on capability; pgvector simpler to invoke
Index routing	Requires an explicit predicate / the UDR; `ORDER BY` alone falls back to a scan	Optimiser-integrated — `ORDER BY <=>` auto-uses the index	pgvector
Intra-query parallelism	No parallel workers within a single UDR scan	Parallel sequential scan for brute force	pgvector
Index maintenance	`hnsw_index_repack()` reclaims tombstoned nodes on demand; `hnsw_am_stats()` reports when it is worthwhile. No background hook.	Automatic VACUUM reclaims bloat	pgvector (automatic)
Engine-version support	**Informix 14 and 15** — identical on-disk format, no data migration between versions	n/a	—
Brute-force query speed	**Parity** with pgvector (AVX2/FMA SIMD)	Reference	Tie
Indexed query (1M × 1,024)	`hnsw_knn_ids()` int8 ≈ 7.36 ms, recall **1.000** (fully resident)	≈ 17.60 ms, recall 0.996 (memory-mapped)	**Informix** (~2.4× faster, higher recall)
Index memory footprint	**int8 codec ≈ 2.57 GB resident** for 1M × 1024-dim (per-vector-scaled, near-lossless, R@10 = 1.000); float32 ≈ 7.2 GB	float32 HNSW ≈ 7.6 GB on disk + page cache (fp16 `halfvec` available separately)	**Informix** (built-in int8 default)
Aggregates (centroid)	`vector_norm(AVG())` ≈ 3× faster	native `avg(vector)`	**Informix**

Aspect	Informix 14	pgvector	Who wins
Bulk INSERT	Text SQLI is slower per row; the **optional binary vector transfer path** (SENDRECV float32, ~1,700 rows/s) closes the gap to pgvector and is byte-identical	Faster by default (pipelined batch)	Tie with binary transfer; else pgvector
Installation	Server-side module installed once by the DBA	`CREATE EXTENSION vector`	pgvector

> The rows below detail each limitation. Net: pgvector wins on ergonomics and packaging (bulk-load is now a tie once the binary vector transfer path is used); Informix wins where it matters for OpenAI-scale RAG — it is the only engine that ANN-indexes 3,072-dim vectors, reaches brute-force parity, and the `hnsw_knn_ids()` UDR answers a 1M-vector indexed query at top recall with the smallest resident footprint measured here.

12.2. Dimension limit

The `sysxtdtypes.maxlen` column is a signed 16-bit integer (`SMALLINT`), capping any variable-length value at 32,767 bytes. For float32: 32,767 / 4 = **8,191 dimensions maximum**. The implementation defaults to `MAXLEN = 12,288` (3,072 dims = `VECTOR_PAGE_SIZE_OPENAI`), matching the OpenAI `text-embedding-3-large` model out of the box. The constant is controlled by `VECTOR_MAXLEN` in `vector.c`.

Notably, pgvector's HNSW index is limited to **2,000 dimensions** even though brute-force scans work up to 16,000. Informix HNSW supports up to 8,191 dims — meaning Informix is the only engine that can build an ANN index over OpenAI `text-embedding-3-large` (3,072 dims). For enterprise models at 4,096 or 8,192 dims, set `VECTOR_MAXLEN = 16384` (requires a 32 KB dbspace page size). For anything above 8,191 dims, pgvector brute-force is the only option.

12.3. No custom operators

PostgreSQL allows `CREATE OPERATOR <=> (FUNCTION = cosine_distance, ...)` so application code can write `embedding <=> query` directly in ORDER BY. Informix only supports the standard arithmetic and relational operator symbols; the mapping from operator to function uses the standard function-call form. pgvector's `<=>`, `<->`, `<+>`, and `<#>` symbols cannot be reproduced in Informix. Application SQL must use the function-call form.

12.4. Index routing requires an explicit predicate

In pgvector, a query like `ORDER BY embedding <=> query LIMIT 10` automatically triggers the HNSW index — the query optimiser recognises the `<=>` operator as an index-scannable predicate and chooses the HNSW access path. No explicit `WHERE` clause is needed.

In Informix, activating the index requires an explicit predicate in the `WHERE` clause — `WHERE vector_near(embedding, query)` — or a direct call to `hnsw_knn_ids()`. An `ORDER BY` on the distance alone does not route the scan through the HNSW index; the `hnsw_knn_ids()` function is the recommended production path.

```
-- pgvector (PostgreSQL): optimizer automatically uses HNSW index
SELECT * FROM documents ORDER BY embedding <=> query LIMIT 10;

-- Informix: explicit predicate required to trigger the HNSW index scan
SELECT FIRST 10 *
  FROM documents
 WHERE vector_near(embedding, query)          -- activates HNSW index
 ORDER BY cosine_distance(embedding, query);  -- exact re-rank within candidates
```

12.5. Informix 14 and 15 support

The same product runs on **Informix 14 and 15**, and the on-disk index format is **identical** across both — an index built on one major version reloads on the other with no data migration. Informix 15 widens the internal row identifier the index uses to fetch candidate vectors for the exact re-rank; the module handles both widths transparently, and the `hnsw_knn_ids()` function remains the recommended production ANN entry point on every version (recall@10 = 1.000, verified by the parity suite on both engines).

12.6. No automatic HNSW index maintenance

pgvector's HNSW index is integrated with PostgreSQL's VACUUM mechanism: dead tuples are removed and index quality is maintained automatically. In this Informix implementation, soft-deleted nodes are excluded from search results (correctness never depends on reclaim) but remain in the graph, **consuming memory and adding dead-node traversal that slows queries**; recall stays intact until the dead-node ratio grows large, at which point the diluted connectivity can start to cost recall too. Reclaiming them is **explicit** rather than automatic: `hnsw_am_stats('idx_name')` reports the live / dead / pending counts, and `hnsw_index_repack('idx_name')` physically reclaims the tombstoned nodes in a single call — no `DROP INDEX / CREATE INDEX` round-trip. It rebuilds the live graph and holds the index exclusively while it runs, so schedule it in a maintenance window after large bulk deletes. There is still no background auto-VACUUM equivalent.

12.7. No parallel query within a single scan

PostgreSQL can execute a sequential scan with multiple parallel workers, each computing distance for a slice of the table. Informix's UDR execution model runs each function invocation in a single virtual processor, without intra-query parallelism for vector functions today.

13. SQL Reference and pgvector Portability

The `vector` type deliberately mirrors pgvector's function names and semantics, so the same SQL runs on both engines with only one substitution: pgvector's operator symbols (`<=>` , `<->` , `<+>` , `<#>`) become the equivalent function call in Informix. This keeps a migration from pgvector a data copy rather than a query rewrite.

Table 9 — Distance and similarity — Informix vs pgvector

Operation	Informix SQL	pgvector (PostgreSQL)
Cosine distance	cosine_distance(v1, v2)	v1 <=> v2
L2 / Euclidean distance	l2_distance(v1, v2)	v1 <-> v2
Inner (dot) product	inner_product(v1, v2)	v1 <#> v2
L1 / Manhattan distance	l1_distance(v1, v2)	v1 <+> v2

Table 10 — Arithmetic, aggregates and utilities — Informix vs pgvector

Operation	Informix SQL	pgvector (PostgreSQL)
Element-wise add / subtract	v1 + v2 / v1 - v2	v1 + v2 / v1 - v2
Element-wise (Hadamard) product	v1 * v2	v1 * v2
Unary negation	-v	—
Sum / centroid aggregate	SUM(v) / AVG(v)	SUM(v) / AVG(v)
Dimension count	vector_dims(v)	vector_dims(v)
Vector norm (length)	vector_norm(v)	vector_norm(v)
Normalise to unit length	vector_normalize(v)	v / vector_norm(v)
Indexed ANN search	hnsw_knn_ids(q, 'idx', k)	ORDER BY col <=> k

14. Glossary — Vector and Similarity-Search Terms

Terms specific to vector search and embeddings, for readers fluent in Informix but new to the AI-vector domain. (Informix/DataBlade terminology is assumed known.)

Table 11 — Vector glossary

Term	Definition
Embedding	A dense numeric vector produced by a neural model that encodes the meaning of an input (text, image, audio). Semantically similar inputs map to nearby points.
Dense vector	A fixed-length array of float32 values where most entries are non-zero (contrast with sparse vectors such as TF-IDF). The `vector` type stores dense float32.
Dimensionality (dims)	The number of components in the vector (e.g. 1,536 or 3,072). Fixed by the embedding model; all vectors in a column share it.
Latent / embedding space	The high-dimensional real-valued space the model maps inputs into. "Meaning" becomes geometric position.
Cosine distance / similarity	Similarity = cosine of the angle between two vectors (1 = identical direction, 0 = orthogonal); distance = 1 − similarity. Magnitude-independent; the default for text embeddings.
L2 / Euclidean distance	Straight-line distance √Σ(a#−b#)². Sensitive to magnitude as well as direction.
Inner product / dot product	Σ(a#·b#). On unit-normalised vectors it equals cosine similarity; used for MIPS.
MIPS	Maximum Inner-Product Search — retrieving the vectors with the largest dot product against a query (recommendation/ranking models).
L1 / Manhattan distance	Σ|a#−b#|. Less sensitive to outliers than L2.
Normalisation / unit vector	Scaling a vector to length 1 (divide by its L2 norm). Pre-normalising lets cosine search use the cheaper inner-product kernel.
Norm	The vector's length, √Σa#². `vector_norm()` returns it.
k-NN (top-k)	k Nearest Neighbours — the k stored vectors closest to a query under a chosen distance metric.
Exact / brute-force search	Compute the distance to every stored vector and sort. Always correct; cost grows linearly with row count.
ANN	Approximate Nearest Neighbour — trades a small amount of recall for sub-linear query time using an index such as HNSW.

Term	Definition
Recall@k	Fraction of the true exact top-k that an approximate query actually returns (1.0 = perfect). The accuracy metric for ANN.
Curse of dimensionality	In high dimensions distances concentrate and axis-aligned structures (B-trees) lose selectivity — why vectors need a graph/space-partitioning index, not a B-tree.
HNSW	Hierarchical Navigable Small World — a layered proximity graph giving logarithmic-time greedy nearest-neighbour search; the ANN index used here and by pgvector.
Small-world graph	A graph where any node reaches any other in few hops via a mix of short- and long-range links — the property HNSW exploits to navigate quickly.
M	HNSW build parameter: max neighbours per node per layer. Higher M → better recall, more memory.
ef_construction	HNSW build parameter: size of the candidate list while inserting. Higher → better graph quality, slower build.
ef_search	HNSW query parameter: size of the candidate list during search. Higher → higher recall, slower query. Tunable per query.
Re-rank	After the index returns approximate candidates, recompute the exact distance on that small set and sort — restores precision over the candidate set.
IVF / IVFFlat	An alternative ANN index that clusters vectors and searches only the nearest clusters. (pgvector offers it; this implementation uses HNSW.)
Quantisation (SQ/PQ)	Compressing vectors to fewer bits to shrink memory/IO at some recall cost. The default int8 codec is a per-vector scalar quantisation (near-lossless here); a float32 codec is also available.
RAG	Retrieval-Augmented Generation — retrieve semantically relevant documents by vector similarity and feed them to an LLM as grounding context.
Hybrid query	A single query combining vector similarity with conventional relational predicates (date, tenant, category) — the core advantage of an in-database vector type.

15. License, Trademarks and Disclaimer

This DataBlade is a **proprietary, commercial software product** owned and published by **Airtool Technologies** (airtool.io). **All rights reserved.** It is licensed, not sold, and its use is governed by a commercial license agreement. No right to use, copy, modify, distribute, sublicense, or create derivative works is granted except as expressly permitted in a written license from Airtool Technologies. Unauthorized reproduction or distribution is prohibited. Contact info@airtool.io for licensing terms.

Purpose of this document. This document is provided for information only. It describes the product as at the date of publication, is subject to change without notice, forms no part of any agreement, and creates no representation, warranty or commitment. The terms governing any use of the software are those of the written license agreement between you and Airtool Technologies.

Warranty. Warranties, if any, are those expressly set out in that agreement. Except as expressly so provided, and to the maximum extent permitted by applicable law, the software is provided **"as is"**, and Airtool Technologies disclaims all other warranties and conditions, whether express, implied or statutory, including the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

Limitation of liability. Except as expressly provided in that agreement, and to the maximum extent permitted by applicable law: neither party is liable for indirect, incidental, special, punitive or consequential damages, or for loss of profits, revenue, data or business, however caused; and each party's total aggregate liability is limited as set out in that agreement. Nothing in this document excludes or limits liability for death or personal injury caused by negligence, for damage to tangible property caused by negligence, for fraud or fraudulent misrepresentation, for wilful misconduct, or for any other liability that cannot be excluded by law.

Trademarks — no affiliation. IBM® and **Informix®** are trademarks or registered trademarks of International Business Machines Corporation; Informix is currently developed and distributed by HCL under license. This project is **independent** and is **not** affiliated with, sponsored by, or endorsed by IBM or HCL. Product names are used solely for identification and interoperability (nominative fair use); no third-party logos are distributed.

Commercial support. Licensing, production support (with SLAs), custom development and consulting are available from **Airtool Technologies** — airtool.io · info@airtool.io. Support entitlements are defined by your commercial license or support agreement. Bundled third-party components retain their own licenses.

Document revision: 2026-07-17. This edition reflects the current HNSW engine — the trimmed-connection on-disk format, transactional pending-delta persistence, on-demand index maintenance (repack and statistics), and support for both Informix 14 and 15.