Informix UDTs

Wic Metadata

Document Version: 1.104 – 2026-06-09

Informix Vector Type

Airtool

Content

This document describes the complete enterprise implementation of a **vector opaque type** for IBM Informix 14.10, including the C data type, AVX2 SIMD-optimised distance kernels, SQL operator overloading, a custom HNSW approximate nearest-neighbour index via the Virtual-Index Interface, and a detailed comparison with the PostgreSQL **pgvector** extension.

The implementation covers the full engineering lifecycle: type registration and de-registration, memory management within the Informix DataBlade API, performance benchmarking, known limitations of the Informix extensibility model relative to pgvector, and production-readiness considerations for Retrieval-Augmented Generation (RAG) and AI-embedding workloads.

1 Build-time configuration

Two compile-time variables control key behaviour of the vector type and its HNSW index. Both are set in the source files and take effect when the shared objects are rebuilt and reinstalled into Informix.

Compile-time configuration variables

Variable	Default	Override
`VECTOR_MAXLEN` (in `vector.c`)	`VECTOR_PAGE_SIZE_OPENAI` = 12,288 bytes → 3,072 max dimensions	Change to any `VECTOR_PAGE_SIZE_*` constant or an explicit byte count up to 32,767, then rebuild
	Effect: Sets the `MAXLEN` clause of `CREATE OPAQUE TYPE vector`. Determines the maximum number of float32 dimensions stored per row.	
`HNSW_STORAGE_DB` (in `hnsw.c`)	1 — database table storage (`hnsw_index_storage` BLOB)	Compile with `-DHNSW_STORAGE_DB=0` to switch to filesystem storage
	Effect: Selects the HNSW persistence backend. DB storage survives server restarts and backup/restore cycles with no extra configuration. File storage writes to `$INFORMIXDIR/extend/hnsw/` and avoids DB overhead but requires write access to that directory.	

VECTOR_MAXLEN: the default is now `VECTOR_PAGE_SIZE_OPENAI` = 12,288 bytes, supporting up to **3,072 float32 dimensions** — matching the OpenAI `text-embedding-3-large` model directly without any custom compile flag. Earlier releases defaulted to `VECTOR_PAGE_SIZE_8K` = 8,184 bytes (2,046 dims). The Informix architectural ceiling is 32,767 bytes = 8,191 dimensions, imposed by the `SMALLINT` type of the `sysxtdtypes.maxlen` column.

HNSW_STORAGE_DB: database storage is now the default (previously required an explicit `-DHNSW_STORAGE_DB` compiler flag). To revert to file storage, compile with `-DHNSW_STORAGE_DB=0`. The on-disk binary format is identical for both backends; only the transport layer differs.

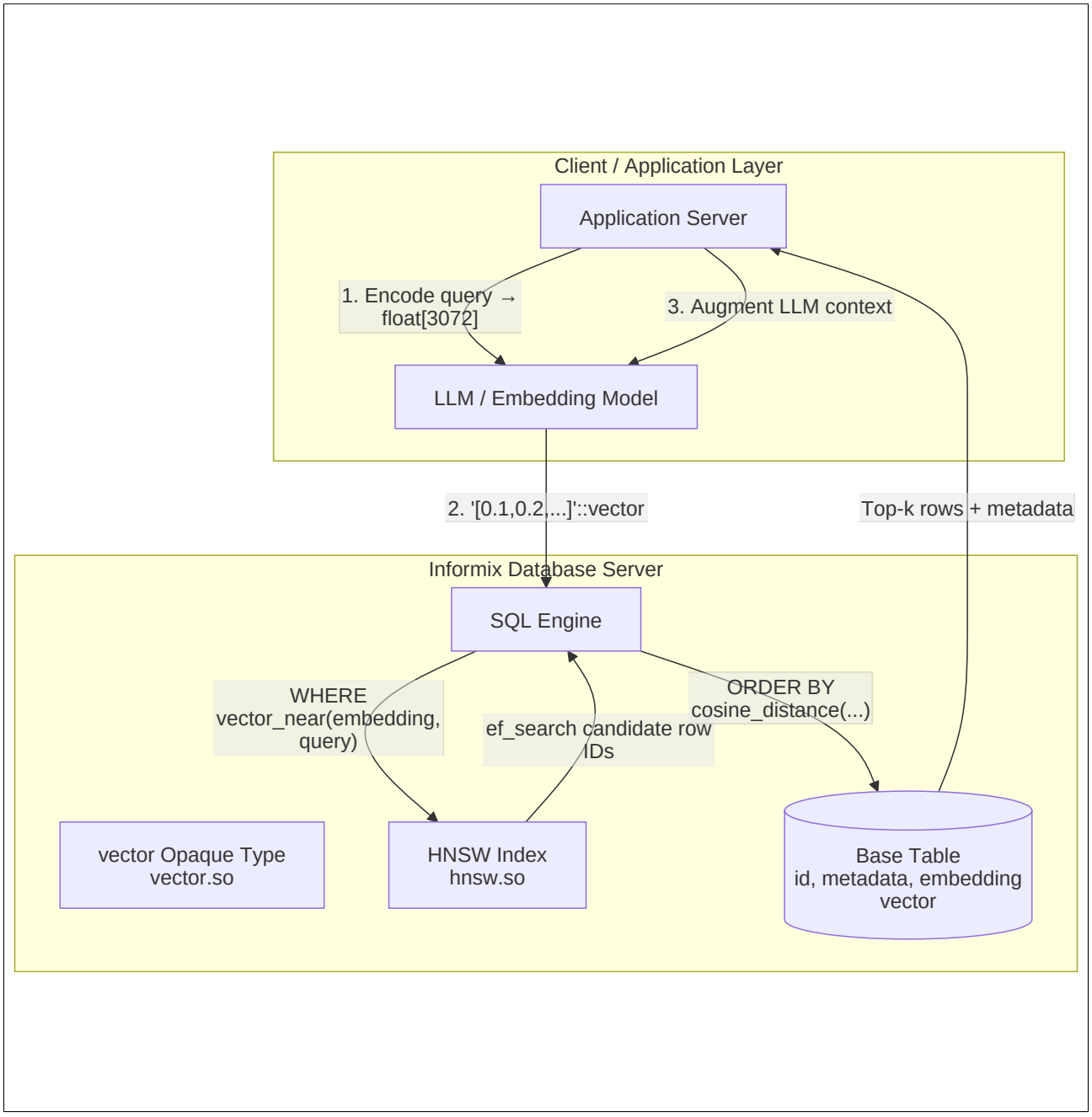
2 Motivation — Vectors in Enterprise Data Platforms

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.

2.1. 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.



3 The vector Opaque Type

Informix allows user-defined data types through its **Opaque Type** mechanism. An opaque type is an extensible binary structure stored in a variable-length `mi_lvarchar` slot. The database server knows nothing about the internal layout; all semantics (parsing, printing, comparison, arithmetic) are provided by C functions compiled into a shared object and registered via DDL.

3.1. Internal representation

The vector type stores a flat array of 32-bit single-precision floats (`mi_real` , IEEE 754 float). The element count (dimension) is recovered at runtime by dividing the stored byte length by the element size:

```
typedef mi_real  vector_t;   /* 32-bit float — matches pgvector's internal type */

dim = mi_get_varlen((mi_lvarchar *) value) / sizeof(vector_t);
```

This layout is identical to pgvector's on-disk representation for `vector(n)` , enabling byte-compatible serialisation between the two engines.

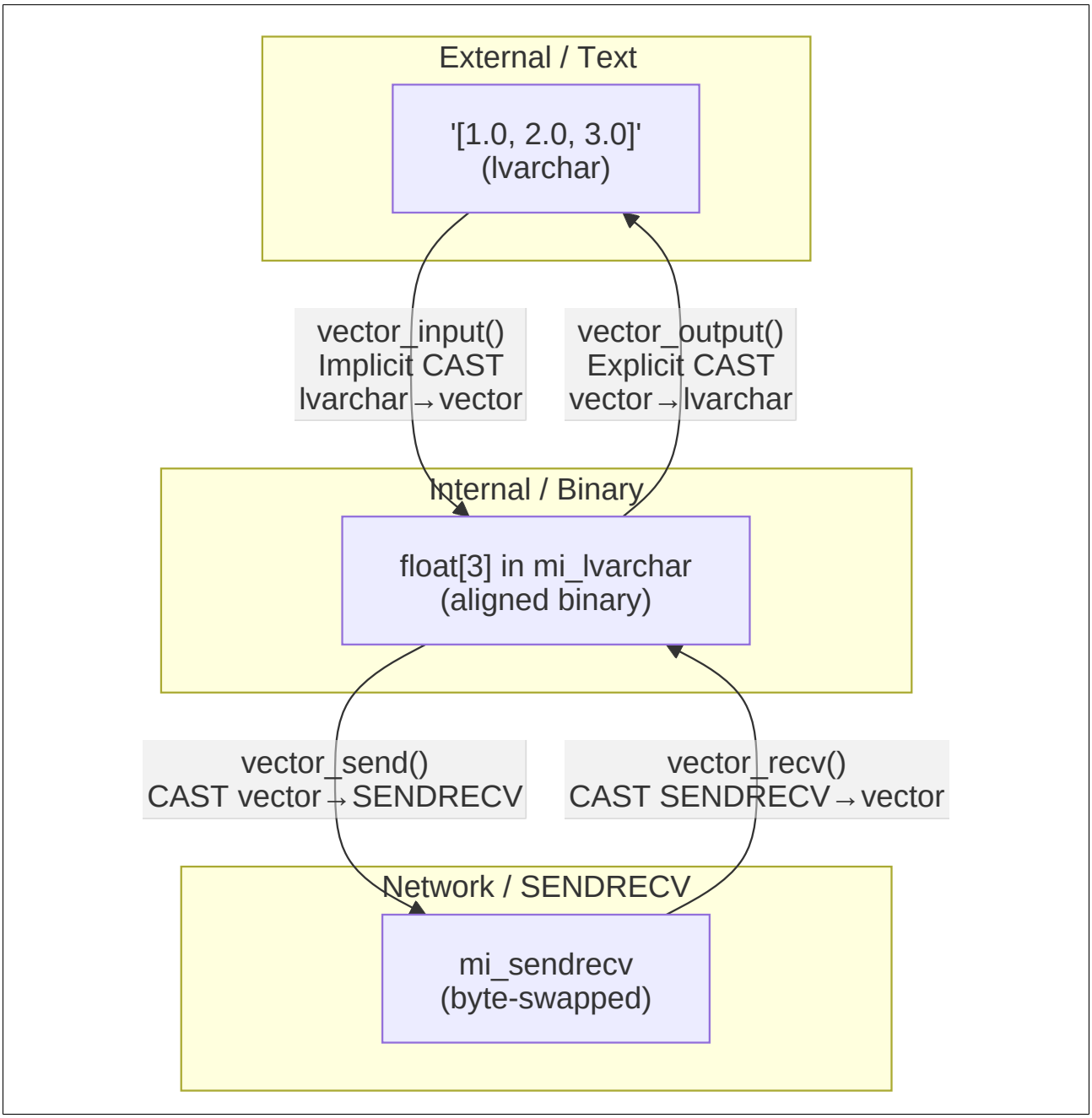
3.1.1. Type declaration

```
-- Variable-length opaque type.  alignment=4 matches float32.
-- MAXLEN caps storage per row: 12,288 bytes / 4 bytes per float = 3,072 dimensions.
-- Controlled by VECTOR_MAXLEN in vector.c (default VECTOR_PAGE_SIZE_OPENAI = 12288
 bytes).
-- The sysxtdtypes.maxlen column is mi_smallint (signed 16-bit), so the absolute
-- ceiling is 32,767 bytes = 8,191 dimensions.

CREATE OPAQUE TYPE vector (INTERNALLENGTH = VARIABLE, alignment = 4, MAXLEN = 12288);
GRANT USAGE ON TYPE vector TO public;
```

> Informix's `MAXLEN` is stored in a `SMALLINT` column (`sysxtdtypes.maxlen`), hard-capping all variable-length opaque types at 32,767 bytes — 8,191 float32 dimensions. pgvector supports up to 16,000 dimensions. This is a hard architectural limit of Informix 14.10.

3.2. Support functions (required by the opaque type mechanism)

Every Informix opaque type must register four support functions that the engine calls implicitly when reading from or writing to network or disk. The table below maps the Informix support function roles to their pgvector equivalents.



3.2.1. Registration SQL

```
-- %s is replaced at install time with the server-side path to vector.so
-- (e.g. /home/informix/extend/studio/test)

CREATE FUNCTION vector(lvarchar)
    RETURNS vector
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_input)'
    LANGUAGE C;

CREATE FUNCTION vector_input(lvarchar)
    RETURNS vector
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so'
    LANGUAGE C;

CREATE FUNCTION vector_output(vector)
    RETURNS lvarchar
```

```
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so'
    LANGUAGE C;

CREATE FUNCTION vector_recv(SENDRECV)
    RETURNS vector
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_recv)'
    LANGUAGE C;

CREATE FUNCTION vector_send(vector)
    RETURNS SENDRECV
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_send)'
    LANGUAGE C;

-- Implicit cast: string literal # vector (enables '[1,2,3]'::vector syntax)
CREATE implicit CAST (lvarchar AS vector WITH vector_input);
CREATE explicit CAST (vector AS lvarchar WITH vector_output);
CREATE implicit CAST (SENDRECV AS vector WITH vector_recv);
CREATE explicit CAST (vector AS SENDRECV WITH vector_send);
```

3.3. Deregistration SQL

Deregistration must drop in dependency order: casts first (which reference the type and its functions), then functions, then the type itself. Functions that reference `vector` as a parameter type must be prefixed with `!` so the SQL executor silently skips them when the type has already been removed.

> The `DROP TYPE vector RESTRICT` statement will fail if any table column still uses the `vector` type. Drop or alter all dependent columns first.

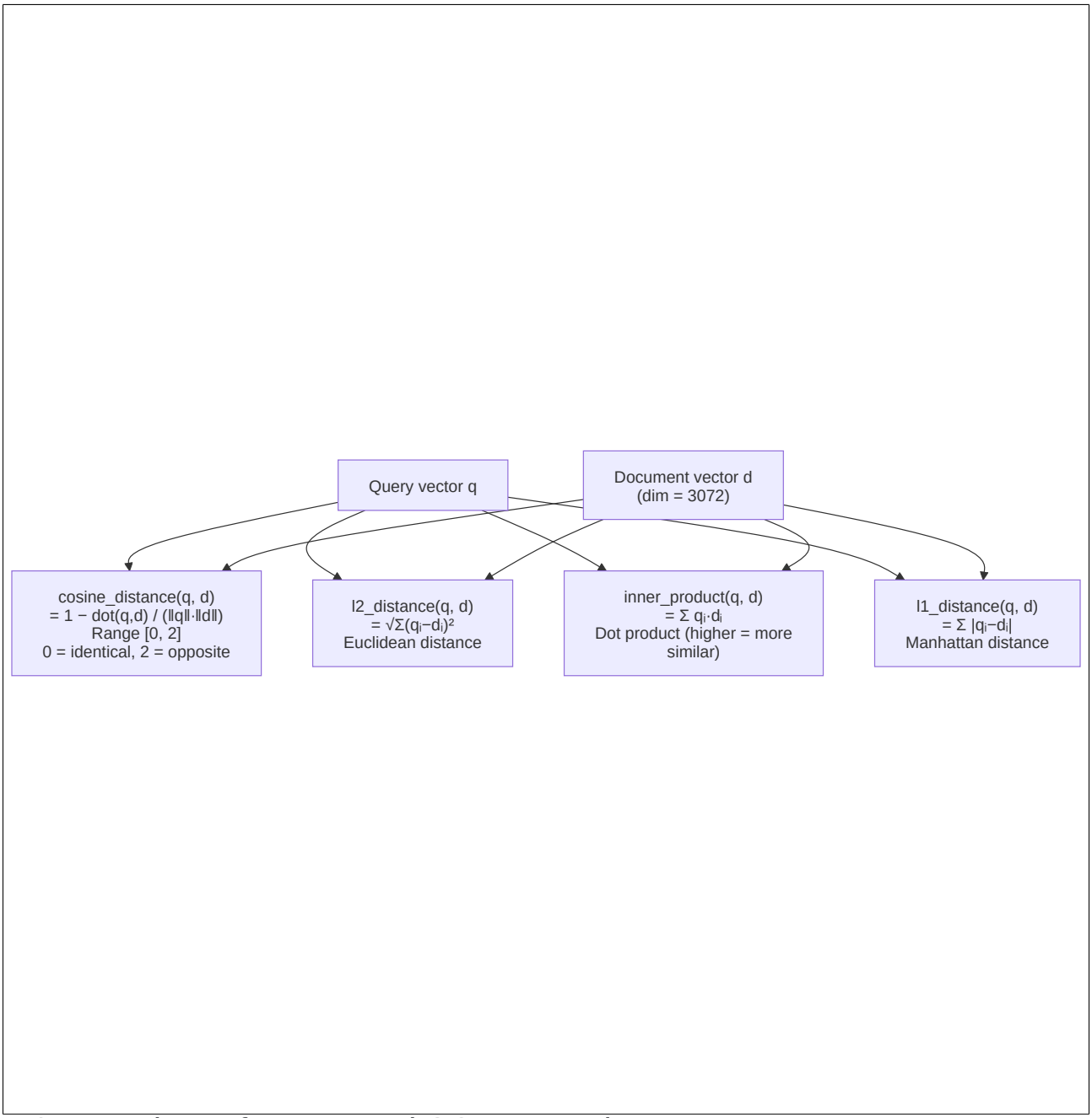
```
-- Casts (! = suppress error if type already missing)
drop cast if exists (lvarchar as vector);
drop cast if exists (vector as lvarchar);
drop cast if exists (vector AS SENDRECV);
drop cast if exists (SENDRECV AS vector);

-- Operator-overloaded functions (! prefix: silently skip if vector type absent)
drop function if exists plus(vector, vector);
drop function if exists minus(vector, vector);
drop function if exists times(vector, vector);
drop function if exists negate(vector);
drop function if exists compare(vector, vector);
drop function if exists divide(vector, integer);

-- Arithmetic functions (names are unique — no parameter type needed)
drop function if exists vector_add;
drop function if exists vector_sub;
drop function if exists vector_mul;

-- Distance / similarity functions
drop function if exists cosine_distance;
drop function if exists l2_distance;
drop function if exists inner_product;
drop function if exists l1_distance;

-- Utility functions
drop function if exists vector_norm;
```

```
drop function if exists vector_normalize;
!drop function if exists vector_dims(vector);
!drop function if exists equal(vector, vector);

-- I/O functions
drop function if exists vector;
drop function if exists vector_input;
drop function if exists vector_output;
drop function if exists vector_send;
drop function if exists vector_recv;

-- Type (fails if any table column still uses vector)
drop type if exists vector RESTRICT;
```

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



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 10 ROWS ONLY;
```

```
-- 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;
```

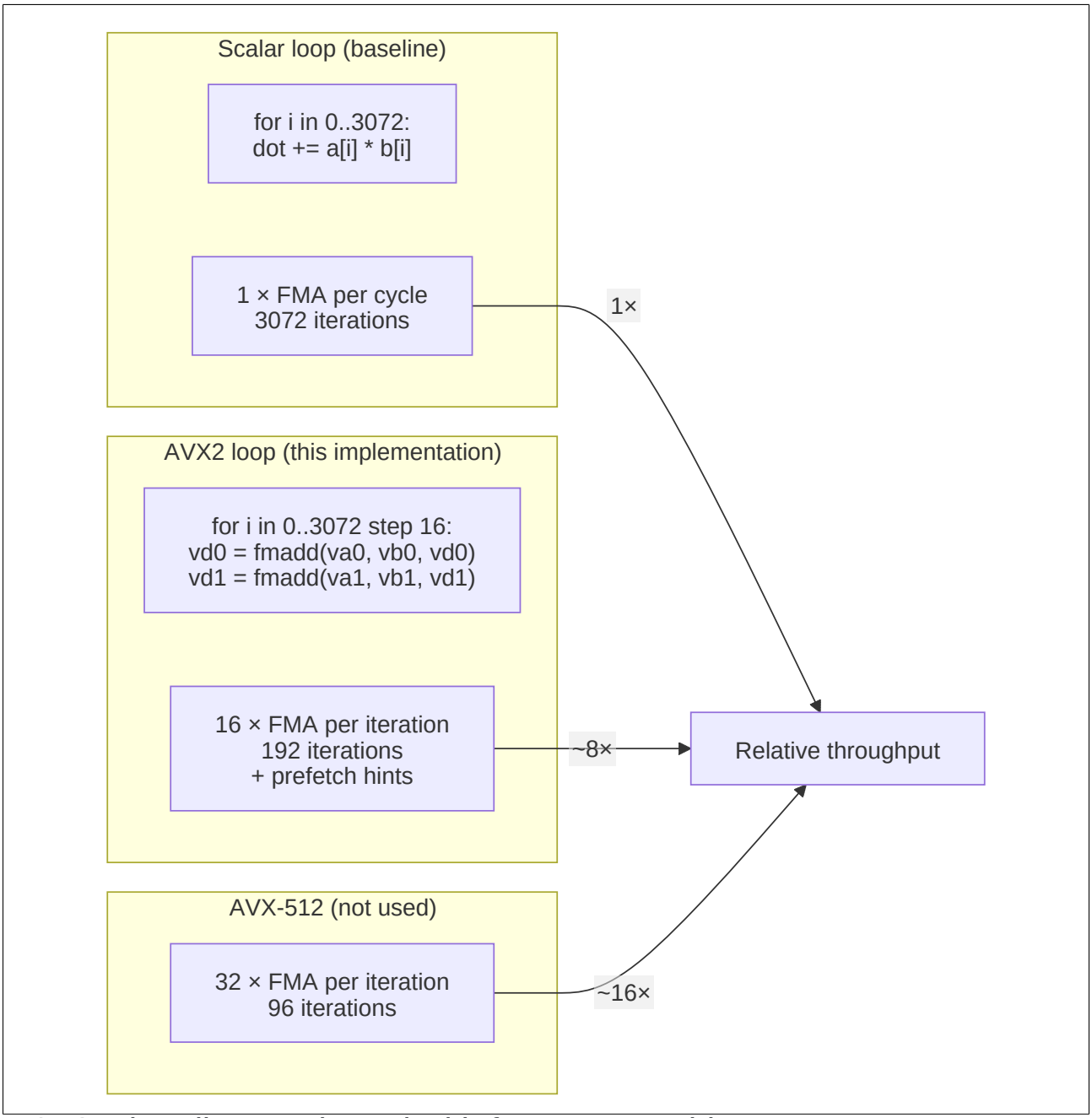
4.3. Registration SQL — distance functions

```
CREATE FUNCTION cosine_distance(arg1 vector, arg2 vector)
    RETURNS FLOAT
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(cosine_distance)'
    LANGUAGE C;

CREATE FUNCTION l2_distance(arg1 vector, arg2 vector)
    RETURNS FLOAT
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(l2_distance)'
    LANGUAGE C;

CREATE FUNCTION inner_product(arg1 vector, arg2 vector)
    RETURNS FLOAT
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(inner_product)'
    LANGUAGE C;

CREATE FUNCTION l1_distance(arg1 vector, arg2 vector)
    RETURNS FLOAT
    WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(l1_distance)'
    LANGUAGE C;
```

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



5.2. Cosine distance kernel with funcstate caching

The cosine distance function implements two orthogonal optimisations:

1. **AVX2 SIMD**: two independent accumulator chains (dual-lane unrolling) to saturate the CPU's multiply-add pipeline and hide memory latency. A prefetch hint pulls the next cache line 32 elements ahead.
2. **Per-command funcstate caching**: in an ANN scan `ORDER BY cosine_distance(col, query)`, the query vector is constant across all rows. The norm² of the query is computed once on the first row invocation and

stored in `MI_FPARAM` funcstate using `mi_dalloc(PER_COMMAND)`. Subsequent rows use the cached value, reducing per-row work from 3N to 2N multiply-accumulate operations (33% fewer FLOPs).

> **Critical DataBlade API constraint:** funcstate memory must be allocated with `mi_dalloc(sizeof(T), PER_COMMAND)`, not `mi_alloc(sizeof(T))`. The default memory duration is `PER_ROUTINE`, meaning the allocation is freed after each individual UDR invocation. Using `mi_alloc` for funcstate creates a dangling pointer on the second row invocation, causing heap corruption that manifests as an `mt_shm_free_blkpool` assertion failure deep inside Informix's shared-memory sort cleanup code — far from the offending UDR, making the root cause extremely difficult to diagnose.

```
/* Per-command funcstate — caches norm²(arg2) across all rows in one SQL command */
typedef struct {
    float norm_q_sq;
} cosine_state_t;

mi_double_precision* cosine_distance(
    mi_bitvarying *Gen_param1,   /* row vector — changes each row */
    mi_bitvarying *Gen_param2,   /* query vector — constant per command */
    MI_FPARAM     *fparam)
{
    vector_t *vec1   = mi_get_vardata_align(Gen_param1, sizeof(vector_t));
    vector_t *vec2   = mi_get_vardata_align(Gen_param2, sizeof(vector_t));
    int       length = mi_get_varlen(Gen_param1) / sizeof(vector_t);

    /* Cache norm²(query) for the full SQL command — computed only on row 1 */
    cosine_state_t *state = (cosine_state_t *) mi_fp_funcstate(fparam);
    if (state == NULL) {
        state = (cosine_state_t *) mi_dalloc(sizeof(cosine_state_t), PER_COMMAND);
        state->norm_q_sq = __norm_sq(vec2, length);   /* AVX2 kernel */
        mi_fp_setfuncstate(fparam, (void *) state);
    }

    float dot, na;
    float nb = state->norm_q_sq;          /* reuse cached value */
    dot = 0.0f; na = 0.0f;
     __dot_and_norma(vec1, vec2, length, &amp;dot, &amp;na);  /* AVX2: dot + norm_a
 in one pass */

    mi_double_precision *result = mi_alloc(sizeof(mi_double_precision));
    mi_double_precision  denom  = sqrt((double)na) * sqrt((double)nb);
    *result = (denom == 0.0) ? 1.0 : 1.0 - ((double)dot / denom);
    return result;
}
```

5.3. AVX2 helper kernels

Three specialised SIMD kernels cover all distance functions. Each uses two independent accumulator chains to exploit instruction-level parallelism, and falls back to scalar code for tail elements below the 8-float SIMD width.

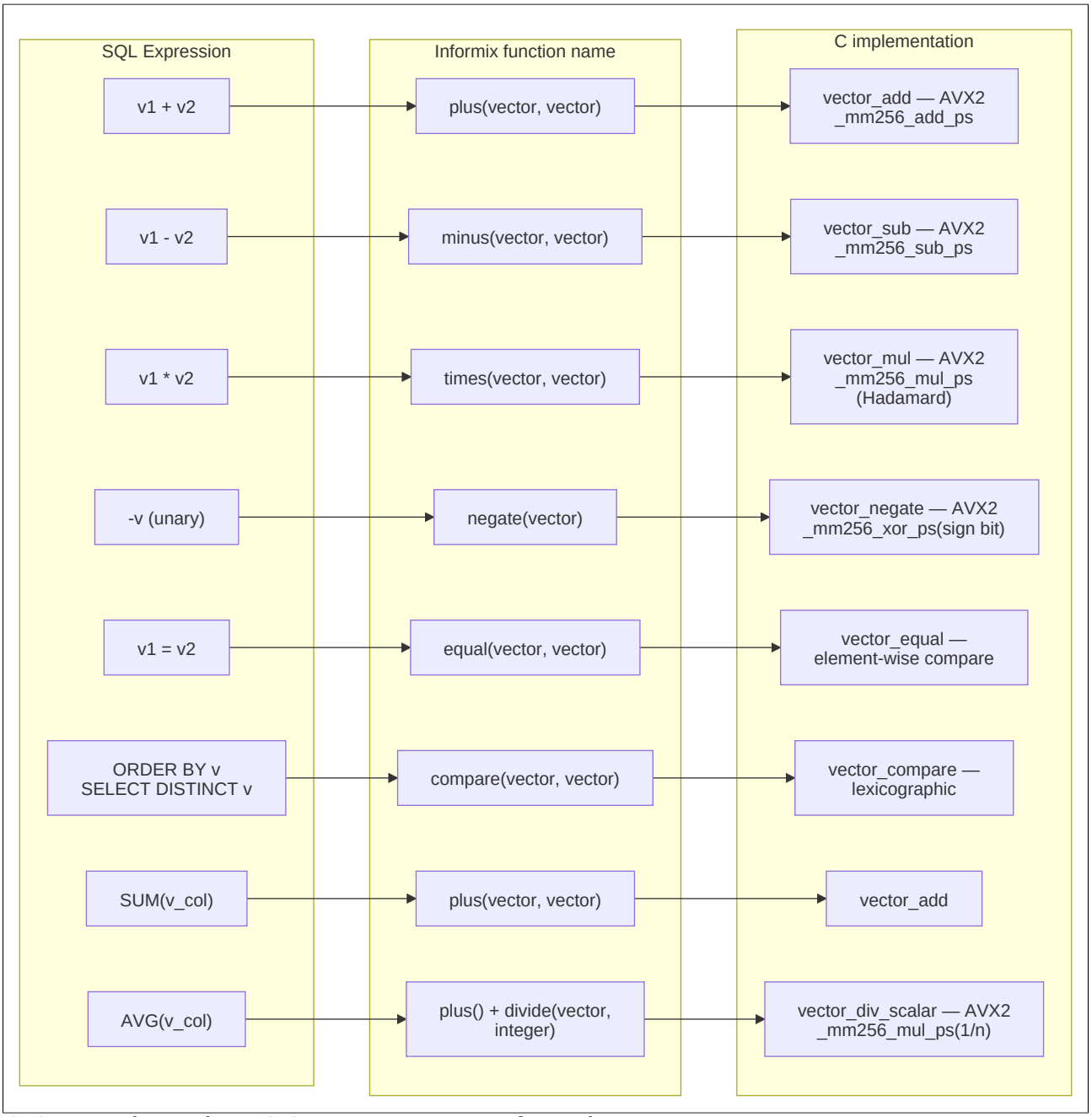
```
/* ## __cosine_accum ############################################################
 * Computes dot(a,b), norm²(a), norm²(b) in a single pass — 3 × 16 floats/iter.
 * Used by l2_distance and inner_product which do not have funcstate caching. */
static inline void __cosine_accum(const float *a, const float *b, int n,
                                  float *out_dot, float *out_na, float *out_nb)
{
    __m256 vd0  = _mm256_setzero_ps(), vd1  = _mm256_setzero_ps();
```

```
    __m256 vna0 = _mm256_setzero_ps(), vna1 = _mm256_setzero_ps();
    __m256 vnb0 = _mm256_setzero_ps(), vnb1 = _mm256_setzero_ps();
    for (int i = 0; i <= n - 16; i += 16) {
          _mm_prefetch((const char *)(a + i + 32), _MM_HINT_T0);   /* prefetch 128
 bytes ahead */
        __m256 va0 = _mm256_loadu_ps(a+i),    va1 = _mm256_loadu_ps(a+i+8);
        __m256 vb0 = _mm256_loadu_ps(b+i),    vb1 = _mm256_loadu_ps(b+i+8);
       vd0  = _mm256_fmadd_ps(va0, vb0, vd0);  vd1  = _mm256_fmadd_ps(va1, vb1, vd1);
      vna0 = _mm256_fmadd_ps(va0, va0, vna0); vna1 = _mm256_fmadd_ps(va1, va1, vna1);
      vnb0 = _mm256_fmadd_ps(vb0, vb0, vnb0); vnb1 = _mm256_fmadd_ps(vb1, vb1, vnb1);
    }
    /* ... 8-float tail, scalar tail, horizontal sum via __hsum256 ... */
}

/* ## __dot_and_norma ############################################################
 * Computes dot(a,b) + norm²(a) in one pass — saves one norm² pass when
 * norm²(b) is already cached in funcstate.  2 × 16 floats/iter. */
static inline void __dot_and_norma(const float *a, const float *b, int n,
                                   float *out_dot, float *out_na)
{
    __m256 vd0  = _mm256_setzero_ps(), vd1  = _mm256_setzero_ps();
    __m256 vna0 = _mm256_setzero_ps(), vna1 = _mm256_setzero_ps();
    for (int i = 0; i <= n - 16; i += 16) {
        _mm_prefetch((const char *)(a + i + 32), _MM_HINT_T0);
        __m256 va0 = _mm256_loadu_ps(a+i),    va1 = _mm256_loadu_ps(a+i+8);
        __m256 vb0 = _mm256_loadu_ps(b+i),    vb1 = _mm256_loadu_ps(b+i+8);
       vd0  = _mm256_fmadd_ps(va0, vb0, vd0);  vd1  = _mm256_fmadd_ps(va1, vb1, vd1);
      vna0 = _mm256_fmadd_ps(va0, va0, vna0); vna1 = _mm256_fmadd_ps(va1, va1, vna1);
    }
    /* ... tail handling ... */
}
```

6 Arithmetic Functions and SQL Operator Overloading

Informix provides **operator binding**: a SQL operator symbol (`+` , `-` , `*`) is automatically mapped to a named function when both operands match registered argument types. There is no `CREATE OPERATOR` statement — registering a function with the exact built-in operator function name (`plus` , `minus` , `times` , …) is sufficient.

6.1. Operator binding map



6.2. Registration SQL — operator functions

```
-- Enables v1 + v2. Also enables SUM(embedding) aggregate.
CREATE FUNCTION plus(arg1 vector, arg2 vector)
    RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_add)' LANGUAGE C;
```

```
-- Enables v1 - v2.
CREATE FUNCTION minus(arg1 vector, arg2 vector)
    RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_sub)' LANGUAGE C;

-- Enables v1 * v2 (Hadamard / element-wise product — NOT dot product).
CREATE FUNCTION times(arg1 vector, arg2 vector)
    RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_mul)' LANGUAGE C;

-- Enables -v (unary negation).
CREATE FUNCTION negate(arg1 vector)
    RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_negate)' LANGUAGE C;

-- Lexicographic comparison — required for ORDER BY, SELECT DISTINCT,
-- GROUP BY, and btree index creation on a vector column.
CREATE FUNCTION compare(arg1 vector, arg2 vector)
    RETURNS INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_compare)' LANGUAGE C;

-- Scalar division: v[i] / n.  Required for AVG(v_col) aggregate.
CREATE FUNCTION divide(arg1 vector, arg2 INTEGER)
    RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_div_scalar)' LANGUAGE C;
```

6.3. 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

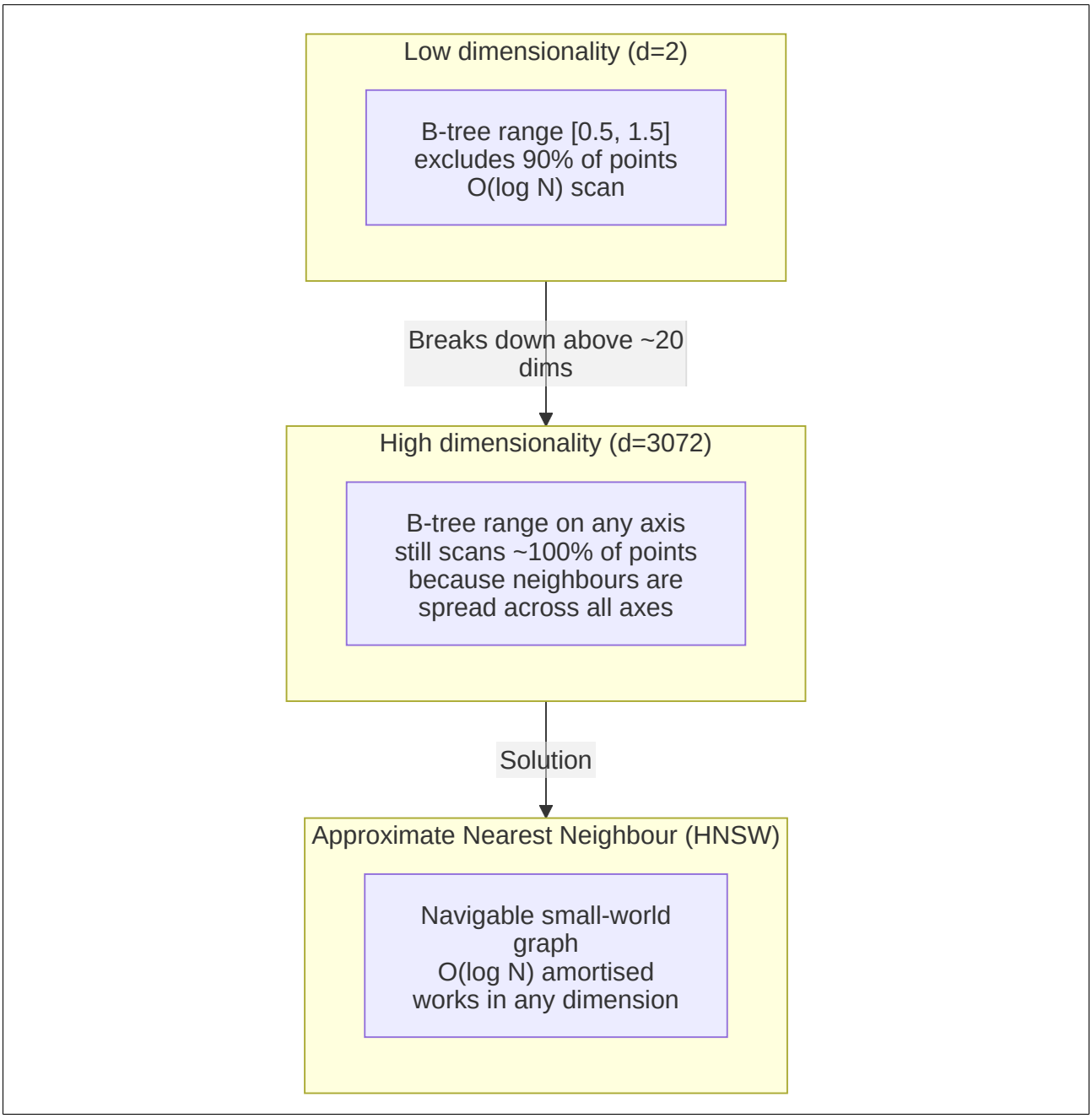
Before examining the HNSW index implementation, it is important to understand why the standard B-tree index — the workhorse of relational database query acceleration — is fundamentally unsuitable for vector similarity search.

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.



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).

This implementation integrates HNSW into Informix as a **Virtual-Index Interface (VTI)** secondary access method, registered via DDL and hooked into the SQL engine through a strategy function predicate.

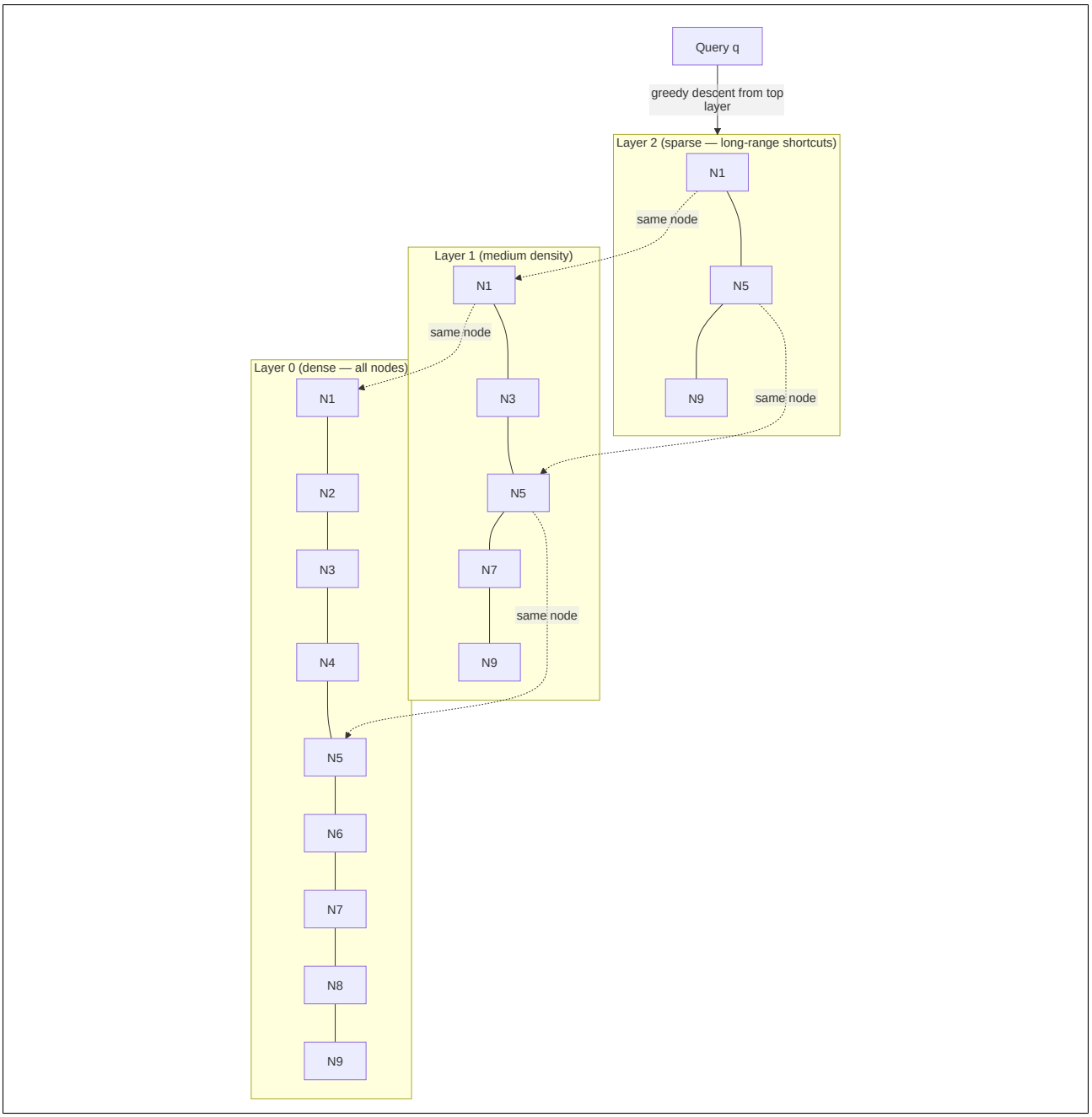
8.1. HNSW algorithm theory

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

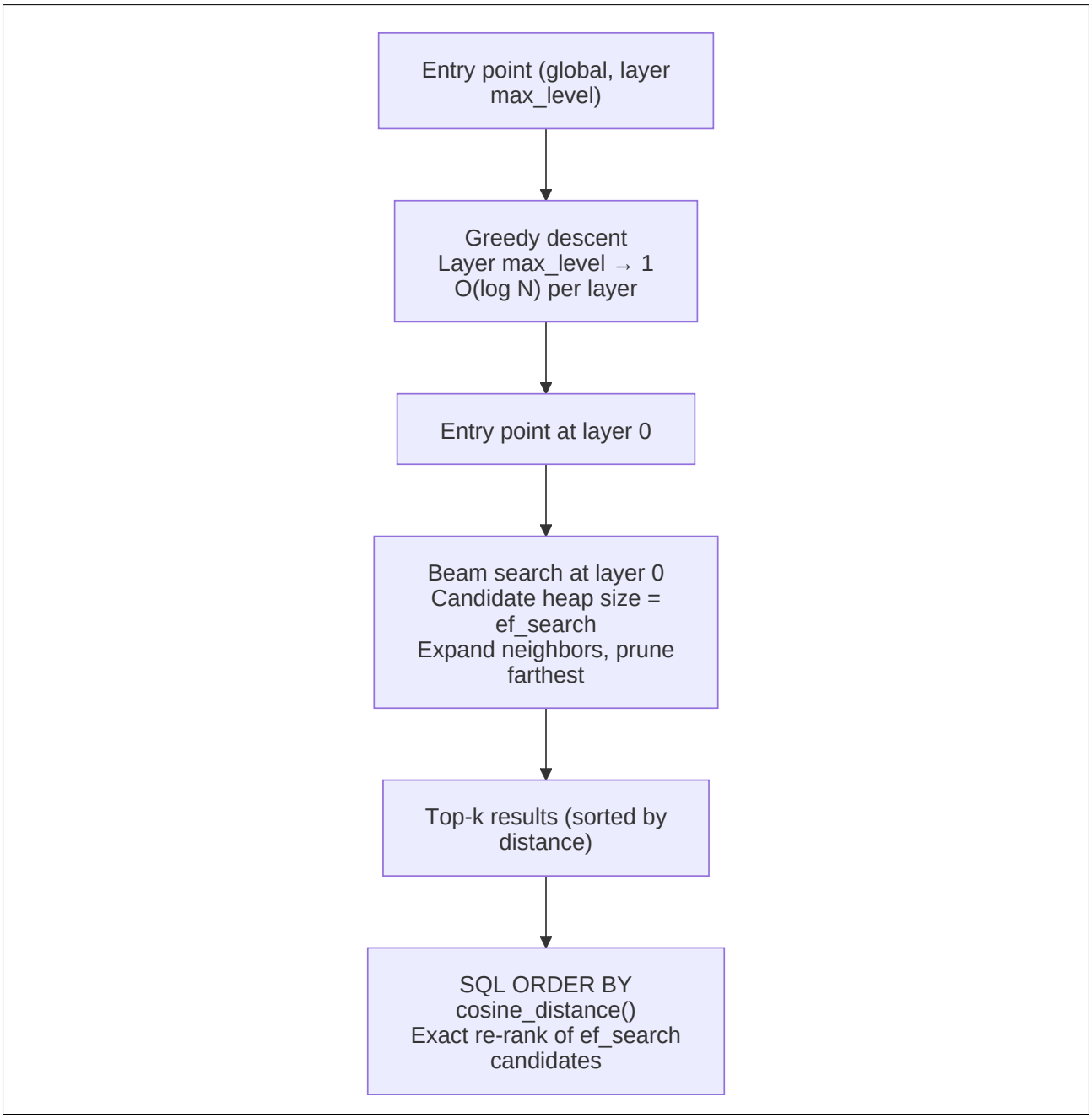
HNSW extends the small-world graph with a **hierarchical structure**. Each node is assigned a random maximum layer at insertion time using an exponential distribution parameterised by M. Layer 0 contains all nodes; higher layers contain progressively fewer nodes, forming a coarse-to-fine hierarchy analogous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



8.1.3. Search algorithm

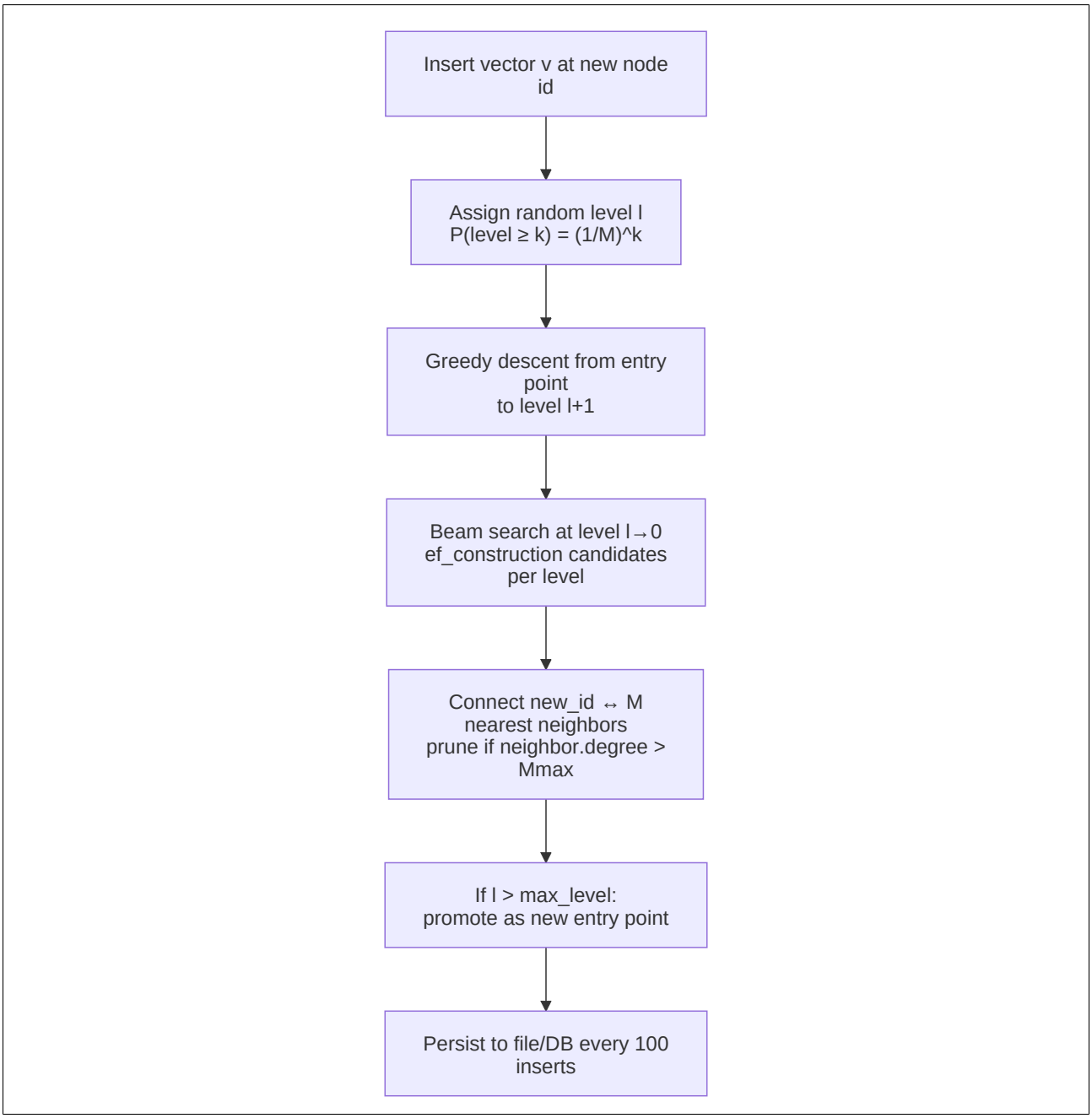
Query traversal proceeds in two phases:

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.



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.



8.2. HNSW parameters

The implementation supports per-index tuning through the index options string. The three parameters control the accuracy/speed trade-off:

```
CREATE INDEX idx_emb ON documents(embedding)
    USING hnsw_am(m='16', ef_construction='200', ef_search='50');
```

- **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 50): beam width during query. Controls the recall/speed trade-off at query time without rebuilding the index. Typical range: 10–200. pgvector default: 40.

8.3. In-memory graph structure (PER_SYSTEM named memory)

The HNSW graph is large — hundreds of megabytes for millions of vectors. Loading it from disk on every query would be catastrophic for latency. The implementation uses Informix **named PER_SYSTEM memory** to keep the entire graph in the shared-memory segment of the Informix server process, persisting across SQL statements and across independent client connections.

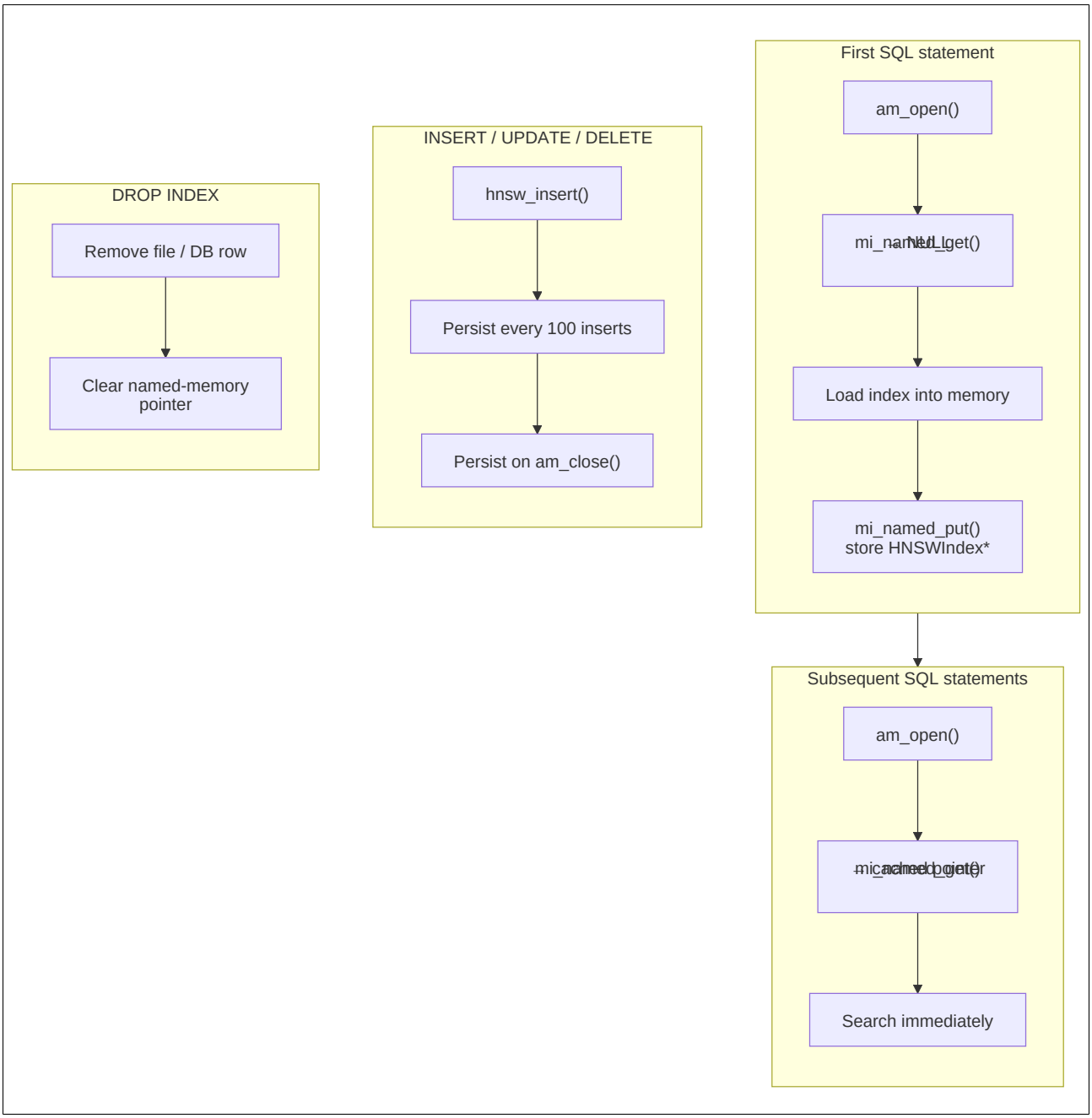
The graph is held in three parallel arrays plus a header struct, all allocated with `mi_dalloc(PER_SYSTEM)`:

PER_SYSTEM in-memory arrays

Array	Type	Size per node	Content
nodes[]	HNSWNode	20 bytes	rowid, fragid, top_level, deleted flag, serial_id
vectors[]	float	dims × 4 bytes	pre-normalised unit-length float32 vectors (flat row-major)
conns[]	mi_integer	HNSW_MAX_LEVELS × (Mmax0+1) × 4 bytes	adjacency lists for all layers; conns[node × 16 × 33 + level × 33 + 0] = neighbour count, [+1..+k] = neighbour node ids

For a typical RAG index of 1 million 3072-dimensional vectors with M=16: nodes = 20 MB, vectors = 12 GB, conns = 2 GB — roughly 14 GB total in shared memory. The arrays grow by doubling capacity (`hnsw_grow`) when the node count reaches the current capacity limit. Old PER_SYSTEM allocations are implicitly retained until server restart; only the active pointer is updated.

The named-memory key is `HNSW_<dbname>__<indexname>`. Both components are sanitised (non-alphanumeric → underscore) so that identical index names in different databases on the same Informix server get distinct slots. Access is serialised with `mi_lock_memory` / `mi_unlock_memory` to prevent concurrent corruption during inserts.



8.4. Storage backends

Two persistence backends are supported, selected at **compile time** by the `HNSW_STORAGE_DB` variable in `hnsw.c`. Both use the same `HNSWFileHeader` binary format; only the transport layer differs.

Storage backend comparison

Aspect	File backend (-DHNSW_STORAGE_DB=0)	DB backend (default)
Location	`$INFORMIXDIR/extend/hnsw/<dbname>__<idx>.hnsw`	Table `hnsw_index_storage` — column `idx_data BLOB`
Write path	`fopen/fwrite` directly to the .hnsw file	serialize to `/tmp/hnsw_<key>.tmp` → `FILETOBLOB(tmp,'server')` → `DELETE + INSERT` → `remove(tmp)`
Read path	`fopen/fread` from the .hnsw file	`mi_lo_open + mi_lo_read` directly into PER_SYSTEM arrays (no temp file)

Aspect	File backend (-DHNSW_STORAGE_DB=0)	DB backend (default)
Drop	`remove(file_path)`	`DELETE FROM hnsw_index_storage WHERE idx_name = key`
Portability	Requires server filesystem access; lost on DB restore	Fully contained in the database; survives backup/restore cycles
Best for	Development, single-server, fastest I/O	All deployments (default); HA/DR, backup/restore portability

> The DB write path must serialize to a `/tmp` file first because the DataBlade API provides no mechanism to stream arbitrary binary data directly into a BLOB column via `mi_exec`. The only standard path is `FILETOBLOB('/path', 'server')`, which reads a file already present on the Informix server's filesystem. The temp file is deleted immediately after the INSERT. An alternative without a temp file (`mi_lo_create + mi_lo_write + UPDATE`) is possible but requires additional DataBlade API surface.

8.4.1. On-disk binary format

Both backends write the same binary layout. The file begins with a fixed 64-byte `HNSWFileHeader`, followed by three flat binary arrays:

```
/* HNSWFileHeader  — 64 bytes, always at offset 0 */
typedef struct {
    mi_integer magic;            /* 0x484E5357 = "HNSW" */
    mi_integer version;          /* 3 = vectors are pre-normalised to unit length */
    mi_integer M;                /* max connections per node per layer */
    mi_integer Mmax0;            /* max connections at layer 0 (= 2*M) */
    mi_integer ef_construction;  /* beam width used during graph build */
    mi_integer ef_search;        /* beam width used during query */
    mi_integer dims;             /* vector dimensionality */
    mi_integer node_count;       /* number of nodes stored */
    mi_integer entry_point;      /* global entry-point node id (-1 if empty) */
    mi_integer max_level;        /* current highest layer */
    float      ml;               /* level generation factor: 1 / ln(M) */
    mi_integer col_idx;          /* column index of vector in base table */
    mi_integer _pad[4];          /* reserved — zero-filled */
} HNSWFileHeader;   /* = 64 bytes */

/* Immediately after the header, three flat arrays follow in sequence:
 *
 *   nodes[node_count]    — HNSWNode structs (20 bytes each)
 *   vectors[node_count]  — node_count × dims float32 values (row-major)
 *   conns[node_count]    — node_count × HNSW_MAX_LEVELS × (Mmax0+1) mi_integers
 *                          conns[n][lv][0]   = neighbour count at layer lv
 *                          conns[n][lv][1..k] = neighbour node ids
 */
```

The `version` field is checked on load and must match `HNSW_VERSION = 3`. A version mismatch causes the load to fail and a fresh empty index to be created, requiring a full index rebuild via `DROP INDEX / CREATE INDEX`.

> Version 3 stores vectors pre-normalised to unit length. This means the on-disk vectors differ from the original application data. Querying with `hnsw_knn_ids()` also normalises the query vector before search, so results are always cosine-based regardless of the original vector magnitudes.

8.4.2. Save strategy

The implementation uses a **write-behind** strategy to balance durability against insert throughput. The graph is persisted in three circumstances:

1. **Periodic flush during bulk inserts:** every `HNSW_SAVE_INTERVAL = 100` inserts, `am_insert` calls the save function. For 10,000-row bulk loads this produces 100 save operations, each rewriting the entire file.
2. **Statement close:** `am_close` is called by Informix at the end of each SQL statement that used the index. If `insert_count > 0` at that point (i.e. the periodic save has not yet fired for the last batch), the graph is saved immediately.
3. **TRUNCATE TABLE:** `am_truncate` resets all graph state to empty and saves the empty header immediately.

> If the Informix server process is killed between periodic saves, up to `HNSW_SAVE_INTERVAL _ 1` (up to 99) inserts may be lost from disk, even though they are present in the in-memory graph. On the next server start the stale on-disk version is loaded. The in-memory graph and on-disk file diverge until the next save fires. Production deployments should reduce `HNSW_SAVE_INTERVAL` or implement WAL-style logging.

8.5. VTI registration SQL

```
-- 1. Register the 12 VTI purpose functions
CREATE FUNCTION hnsw_am_create(pointer)  RETURNING INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_create)'  LANGUAGE C;
CREATE FUNCTION hnsw_am_drop(pointer)    RETURNING INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_drop)'    LANGUAGE C;
CREATE FUNCTION hnsw_am_open(pointer)    RETURNING INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_open)'    LANGUAGE C;
CREATE FUNCTION hnsw_am_close(pointer)   RETURNING INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_close)'   LANGUAGE C;
CREATE FUNCTION hnsw_am_insert(pointer, pointer, pointer) RETURNING INTEGER WITH (NOT
 VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_insert)'  LANGUAGE C;
CREATE FUNCTION hnsw_am_delete(pointer, pointer, pointer) RETURNING INTEGER WITH (NOT
 VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_delete)'  LANGUAGE C;
CREATE FUNCTION hnsw_am_update(pointer, pointer, pointer, pointer, pointer) RETURNING
 INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_update)'  LANGUAGE C;
CREATE FUNCTION hnsw_am_beginscan(pointer) RETURNING INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_beginscan)' LANGUAGE C;
CREATE FUNCTION hnsw_am_getnext(pointer, pointer, pointer) RETURNING INTEGER WITH (NOT
 VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_getnext)' LANGUAGE C;
CREATE FUNCTION hnsw_am_endscan(pointer)  RETURNING INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_endscan)' LANGUAGE C;
CREATE FUNCTION hnsw_am_scancost(pointer, pointer) RETURNING POINTER WITH (NOT VARIANT)
```

```
    EXTERNAL NAME '%s/hnsw.so(am_scancost)' LANGUAGE C;
CREATE FUNCTION hnsw_am_truncate(pointer) RETURNING INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(am_truncate)' LANGUAGE C;

-- 2. Create the secondary access method
CREATE SECONDARY ACCESS_METHOD hnsw_am
    (AM_OPEN     = hnsw_am_open,   AM_CLOSE    = hnsw_am_close,
     AM_CREATE   = hnsw_am_create, AM_DROP     = hnsw_am_drop,
     AM_BEGINSCAN = hnsw_am_beginscan, AM_GETNEXT  = hnsw_am_getnext,
     AM_ENDSCAN  = hnsw_am_endscan, AM_INSERT   = hnsw_am_insert,
     AM_DELETE   = hnsw_am_delete, AM_UPDATE   = hnsw_am_update,
     AM_SCANCOST = hnsw_am_scancost, AM_TRUNCATE = hnsw_am_truncate,
     AM_SPTYPE   = 'A');

-- 3. Strategy function (WHERE predicate hook)
CREATE FUNCTION vector_near(vector, vector) RETURNS BOOLEAN WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(vector_near)' LANGUAGE C;

-- 4. Support function (required by OPCLASS SUPPORT clause)
CREATE FUNCTION vector_cmp(vector, vector) RETURNS INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/hnsw.so(vector_cmp)' LANGUAGE C;

-- 5. Operator class + set as default for the access method
CREATE OPCLASS hnsw_vector_ops FOR hnsw_am
    STRATEGIES (vector_near)
    SUPPORT (vector_cmp);

ALTER ACCESS_METHOD hnsw_am ADD AM_DEFOPCLASS = hnsw_vector_ops;

-- 6. Direct ANN function (bypasses VTI routing — used by benchmarks)
CREATE FUNCTION hnsw_knn_ids(vector, LVARCHAR, INTEGER) RETURNING LVARCHAR WITH (NOT
 VARIANT)
    EXTERNAL NAME '%s/hnsw.so(hnsw_knn_ids)' LANGUAGE C;
```

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='50');

-- ANN search via HNSW index
-- vector_near() is the WHERE-clause hook that triggers the VTI scan.
-- am_beginscan() pre-computes ef_search candidate row IDs via HNSW traversal.
-- am_getnext() returns candidates one at a time to the outer query.
-- ORDER BY cosine_distance() re-ranks exactly within the candidate set.
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 without VTI routing (benchmark / diagnostics)
SELECT hnsw_knn_ids('[0.1, 0.2, ...]'::vector, 'idx_emb', 10);
```

8.7. Pre-normalisation optimisation

The HNSW implementation pre-normalises every inserted vector to unit length (`normalize_vec()` called inside `hnsw_insert()`). For unit-length vectors, the general cosine distance formula simplifies dramatically:

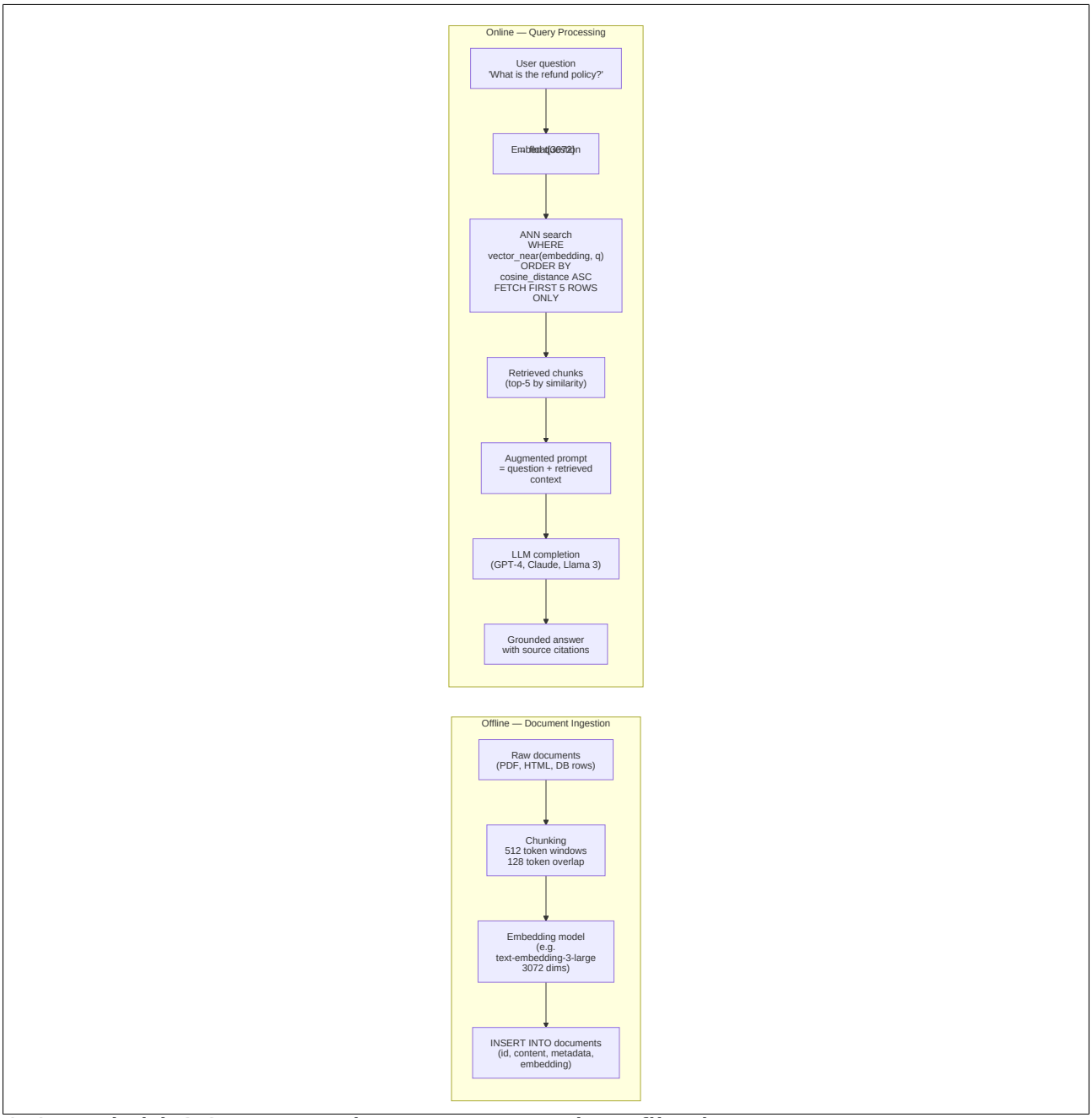
```
/* General cosine distance (3 passes over the data):
 *   1 - dot(a,b) / (sqrt(norm_sq(a)) * sqrt(norm_sq(b)))
 *
 * With pre-normalisation both norms are exactly 1.0, so the kernel reduces to:
 *   1 - dot(a, b)                 (1 pass, no sqrt, no division)
 */
static float cosine_dist(const float *a, const float *b, int n)
{
    float dot = 0.0f;
    for (int i = 0; i < n; i++) dot += a[i] * b[i];
    return 1.0f - dot;
}
```

This eliminates two `sqrt()` calls and two norm accumulations per distance evaluation during graph search. For 3072-dimensional vectors with ef_search = 50 and a graph of 1M nodes, this saves roughly **600M floating-point operations per query**.

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



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 Performance Benchmarks — Informix vs pgvector

Benchmarks compare Informix 14.10 (AVX2-optimised UDR) against PostgreSQL 16 + pgvector 0.7 at the OpenAI `text-embedding-3-large` dimension of **3,072 dims**. (10 timed queries, 2 warm-up rounds, ground truth = exact brute-force top-10). HNSW parameters: **m=16, ef_construction=200, ef_search=50** on both engines where applicable.

> **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.

10.1. AVX2 optimisation (2,046-dim benchmark)

Measurements at 1,000 rows × **2,046 dimensions** showing the impact of adding explicit AVX2 SIMD intrinsics.

Informix vs PostgreSQL — scalar C loops vs AVX2 SIMD (1K rows × 2046 dims)

Optimisation	Operation	top-k	IFX ms/q	PG ms/q	IFX / PG
Scalar (before)	INSERT total	1,000	7,307	5,514	1.33x
Scalar (before)	cosine	10	12	3	4.00x
Scalar (before)	L2	1	12	4	3.00x
AVX2 (after)	INSERT total	1,000	5,559	5,711	0.97x
AVX2 (after)	cosine	10	9	3	3.00x
AVX2 (after)	L2	10	9	3	3.00x
AVX2 (after)	inner product	10	8	4	2.00x

> AVX2 improvements: cosine top-10 ratio fell from 4.00× to 3.00×; INSERT flipped — IFX became marginally faster (0.97×) because UDR dispatch overhead is lower than PostgreSQL parse/plan overhead when computing no distance functions.

10.2. Head-to-head comparison — 1,536 dims (text-embedding-3-small, both engines HNSW-capable)

At 1,536 dimensions both pgvector and Informix can build HNSW indexes, enabling a fair apples-to-apples comparison for all operation categories. Measurements from HNSW parameters: m=16, ef_construction=200, ef_search=50. Ground truth for recall: each engine's own brute-force cosine top-10.

PG pgvector vs IFX vector UDT — all operations, 1K rows × 1536 dims

Operation	PG (BF)	IFX (BF)	IFX/PG (BF)	PG HNSW	IFX HNSW	IFX/PG (HNSW)
INSERT 1,000 rows	5,755 ms	7,157 ms	1.24x	—	—	—
cosine top-10 (ms/q)	4 ms	8 ms	2.00x	4 ms	6 ms	1.50x
L2 top-10 (ms/q)	4 ms	8 ms	2.00x	—	—	—
inner_product top-10 (ms/q)	4 ms	7 ms	1.75x	—	—	—
L1 top-10 (ms/q)	4 ms	8 ms	2.00x	—	—	—
HNSW build time	—	—	—	488 ms	1,836 ms	3.76x
Recall@10 (cosine)	—	—	—	100%	89%	—
HNSW vs BF speedup	—	—	—	1.0x	1.3x	—

PG pgvector vs IFX vector UDT — all operations, 10K rows × 1536 dims

Operation	PG (BF)	IFX (BF)	IFX/ PG (BF)	PG HNSW	IFX HNSW	IFX/PG (HNSW)
INSERT 10,000 rows	64,217 ms	72,523 ms	1.13x	—	—	—
cosine top-10 (ms/q)	26 ms	38 ms	1.46x	5 ms	7 ms	1.40x
L2 top-10 (ms/q)	26 ms	39 ms	1.50x	—	—	—
inner_product top-10 (ms/q)	26 ms	40 ms	1.54x	—	—	—
L1 top-10 (ms/q)	27 ms	38 ms	1.41x	—	—	—
HNSW build time	—	—	—	19,451 ms	42,975 ms	2.21x
Recall@10 (cosine, ef_search=50)	—	—	—	29%	30%	—
HNSW vs BF speedup	—	—	—	5.2x	5.4x	—

Key findings at 1,536 dims:

- **Brute-force:** Informix is 1.41–2.00× slower than pgvector across all metrics. The gap is consistent across cosine, L2, inner_product, and L1.
- **HNSW query latency:** both engines deliver comparable indexed query times (PG 4–5 ms/q vs IFX 6–7 ms/q), 1.40x gap. For a production RAG system, both are sub-10 ms at 10K rows.
- **HNSW build:** pgvector builds the index 2.2–3.8× faster. This matters for bulk-load scenarios but has no impact on steady-state query throughput.
- **Recall at ef_search=50:** both engines achieve identical recall (~29–30%) at 10K rows. Increase ef_search to 100–200 for production accuracy (R@10 $\geq$ 90%).
- **pgvector HNSW advantage:** at 1K rows, perfect recall (100%) vs IFX 89%. pgvector's native HNSW has a more mature graph construction algorithm for small datasets.

10.3. IFX HNSW vs IFX brute-force — 1,000 rows × 3,072 dims

pgvector cannot index 3,072-dim vectors with HNSW (2,000-dim ceiling), so this section focuses solely on Informix: sequential scan vs HNSW-indexed cosine search. For an engine-to-engine HNSW comparison, see the 1,536-dim section above.

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.

10.4. IFX HNSW vs IFX brute-force — 10,000 rows × 3,072 dims

At 10K rows the sequential scan cost grows linearly while HNSW stays sub-linear. pgvector still cannot index at this dimension, so Informix HNSW is the only ANN option for production OpenAI embeddings (text-embedding-3-large).

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

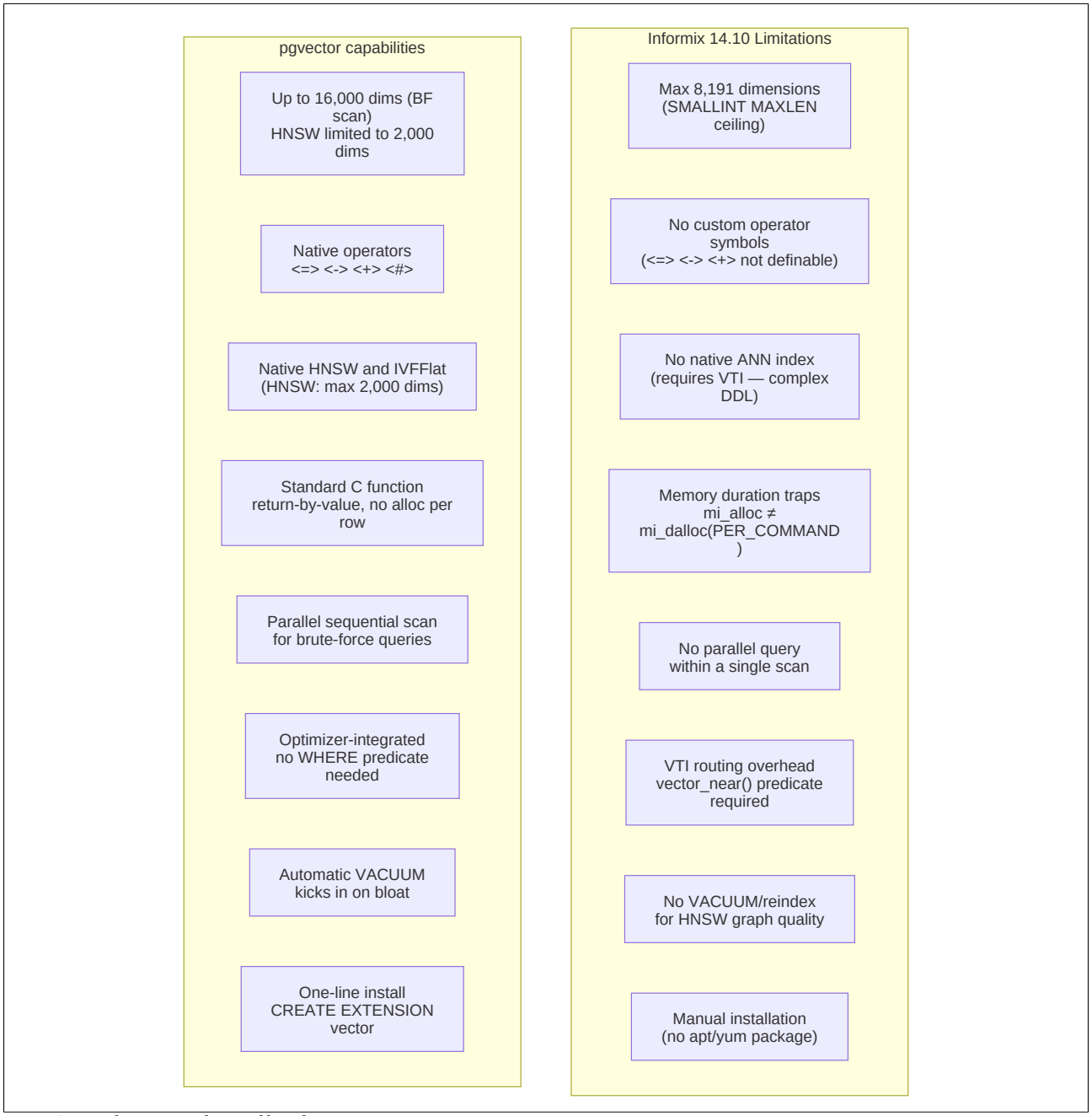
Operation	IFX brute-force	IFX HNSW
HNSW build time	—	86,630 ms
Recall@10 (ef_search=50)	—	27%
HNSW speedup vs brute-force	—	115×

ef_search=50 gives low recall at 10K rows. With 10K nodes, ef_search=50 explores only 0.5% of the graph, yielding R@10=27%. For production RAG, raise ef_search to 100–200 to reach R@10 above 90%. Even at R@10=27%, the 115× speedup over brute-force makes HNSW mandatory at this scale.

11 Informix Limitations vs pgvector

The following limitations were discovered during implementation and testing. They are inherent to the Informix 14.10 extensibility model and cannot be worked around without changes to the database engine itself.

11.1. Limitation summary table



11.2. Dimension limit

The `sysxtdtypes.maxlen` column is a signed 16-bit integer (`SMALLINT`), capping any variable-length opaque type 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.

11.3. 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 is fixed by name convention (Section 5). pgvector's `<=>`, `<->`, `<+>`, and `<#>` symbols cannot be reproduced in Informix. Application SQL must use the function-call form.

11.4. Memory duration traps in the DataBlade API

The DataBlade API has a memory duration system (`PER_ROUTINE`, `PER_COMMAND`, `PER_SYSTEM`) that does not exist in PostgreSQL's C extension model. The default duration for `mi_alloc()` is `PER_ROUTINE` — memory is freed after each individual UDR invocation.

Using `mi_alloc()` to allocate funcstate storage creates a dangling pointer on the second row invocation. The resulting heap corruption manifests as an `mt_shm_free_blkpool` assertion failure in Informix's shared-memory sort cleanup code — far from the offending UDR in the call stack, making root-cause analysis extremely difficult. The correct allocation is `mi_dalloc(size, PER_COMMAND)`.

```
/* WRONG — mi_alloc defaults to PER_ROUTINE, freed after row 1 */
state = (cosine_state_t *) mi_alloc(sizeof(cosine_state_t));

/* CORRECT — PER_COMMAND lifetime spans all rows in the SQL command */
state = (cosine_state_t *) mi_dalloc(sizeof(cosine_state_t), PER_COMMAND);
```

11.5. VTI routing requires a 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's VTI model, index activation requires an explicit strategy-function predicate in the `WHERE` clause: `WHERE vector_near(embedding, query)`. Without this predicate the optimiser does not route the scan through the HNSW access method.

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

-- Informix: explicit predicate required to trigger VTI 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
```

11.6. 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 but remain in the graph, consuming memory and slightly degrading recall. A manual rebuild (`DROP INDEX / CREATE INDEX`) is required to reclaim space after large bulk deletes.

11.7. 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 UDRs. The `PARALLELIZABLE` modifier was tested and caused crashes (likely due to shared state in the funcstate cache), and was reverted.

12 Complete SQL Function Reference

12.1. Full function table

The following table is the authoritative reference for all functions provided by `vector.so`. The pgvector column shows the equivalent expression on PostgreSQL for migration and cross-platform SQL portability.

Utility Functions

C function	Math	pgvector (PostgreSQL)	Informix SQL
vector_dims	dim(v)	vector_dims(v::vector)	vector_dims(v)
vector_equal	v1[i] == v2[i] for all i	v1 = v2	equal(v1,v2) / v1 = v2
vector_norm	sqrt(sum(v[i]²))	vector_norm(v::vector)	vector_norm(v)
vector_normalize	v / |v|	v / vector_norm(v)	vector_normalize(v)

Distance / Similarity Functions

C function	Math	pgvector (PostgreSQL)	Informix SQL
cosine_distance	1 _ dot(v1,v2) / (|v1| · |v2|)	v1 <=> v2 / cosine_distance(v1,v2)	cosine_distance(v1, v2)
l2_distance	sqrt(sum((v1[i]#v2[i])²))	v1 <-> v2 / l2_distance(v1, v2)	l2_distance(v1, v2)
inner_product	sum(v1[i] · v2[i])	inner_product(v1, v2)	inner_product(v1, v2)
l1_distance	sum(|v1[i] _ v2[i]|)	v1 <+> v2 / l1_distance(v1, v2)	l1_distance(v1, v2)

Arithmetic Functions

C function	Math	pgvector (PostgreSQL)	Informix SQL
vector_add	v1[i] + v2[i]	v1 + v2	vector_add(v1, v2)
vector_sub	v1[i] _ v2[i]	v1 _ v2	vector_sub(v1, v2)
vector_mul	v1[i] · v2[i] (Hadamard)	v1 * v2	vector_mul(v1, v2)
vector_negate	#v[i]	—	negate(v)
vector_div_scalar	v[i] / n	—	divide(v, n::INTEGER)

Operator-Overloaded Functions

SQL expression	Informix function name	C implementation	Note
v1 + v2	plus(vector, vector)	vector_add	
v1 _ v2	minus(vector, vector)	vector_sub	
v1 * v2	times(vector, vector)	vector_mul	Hadamard (element-wise)
#v (unary)	negate(vector)	vector_negate	
v1 = v2	equal(vector, vector)	vector_equal	already registered above

SQL expression	Informix function name	C implementation	Note
ORDER BY v / DISTINCT	compare(vector, vector)	vector_compare	lexicographic order
SUM(v_col)	plus(vector, vector)	vector_add	automatic via plus()
AVG(v_col)	plus() + divide(v, integer)	vector_div_scalar	centroid computation

12.2. Complete installation SQL

```
-- ============================================================================
-- vector.so complete installation
-- (replace %s with the server-side directory containing vector.so)
-- ============================================================================

CREATE OPAQUE TYPE vector (INTERNALLENGTH = VARIABLE, alignment = 4, MAXLEN = 12288);
GRANT USAGE ON TYPE vector TO public;

-- I/O support functions
CREATE FUNCTION vector(lvarchar) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_input)' LANGUAGE C;
CREATE FUNCTION vector_input(lvarchar) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so' LANGUAGE C;
CREATE FUNCTION vector_output(vector) RETURNS lvarchar WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so' LANGUAGE C;
CREATE FUNCTION vector_recv(SENDRECV) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_recv)' LANGUAGE C;
CREATE FUNCTION vector_send(vector) RETURNS SENDRECV WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_send)' LANGUAGE C;

-- Utility
CREATE FUNCTION vector_dims(arg1 vector) RETURNING SMALLINT WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_dims)' LANGUAGE C;
CREATE FUNCTION equal(arg1 vector, arg2 vector) RETURNING BOOLEAN WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_equal)' LANGUAGE C;

-- Distance functions
CREATE FUNCTION cosine_distance(arg1 vector, arg2 vector) RETURNS FLOAT WITH (NOT
 VARIANT)
    EXTERNAL NAME '%s/vector.so(cosine_distance)' LANGUAGE C;
CREATE FUNCTION l2_distance(arg1 vector, arg2 vector) RETURNS FLOAT WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(l2_distance)' LANGUAGE C;
CREATE FUNCTION inner_product(arg1 vector, arg2 vector) RETURNS FLOAT WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(inner_product)' LANGUAGE C;
CREATE FUNCTION l1_distance(arg1 vector, arg2 vector) RETURNS FLOAT WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(l1_distance)' LANGUAGE C;

-- Arithmetic
CREATE FUNCTION vector_norm(arg1 vector) RETURNS FLOAT WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_norm)' LANGUAGE C;
CREATE FUNCTION vector_normalize(arg1 vector) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_normalize)' LANGUAGE C;
CREATE FUNCTION vector_add(arg1 vector, arg2 vector) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_add)' LANGUAGE C;
CREATE FUNCTION vector_sub(arg1 vector, arg2 vector) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_sub)' LANGUAGE C;
CREATE FUNCTION vector_mul(arg1 vector, arg2 vector) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_mul)' LANGUAGE C;

-- Operator binding
CREATE FUNCTION plus(arg1 vector, arg2 vector) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_add)' LANGUAGE C;
```

```
CREATE FUNCTION minus(arg1 vector, arg2 vector) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_sub)' LANGUAGE C;
CREATE FUNCTION times(arg1 vector, arg2 vector) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_mul)' LANGUAGE C;
CREATE FUNCTION negate(arg1 vector) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_negate)' LANGUAGE C;
CREATE FUNCTION compare(arg1 vector, arg2 vector) RETURNS INTEGER WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_compare)' LANGUAGE C;
CREATE FUNCTION divide(arg1 vector, arg2 INTEGER) RETURNS vector WITH (NOT VARIANT)
    EXTERNAL NAME '%s/vector.so(vector_div_scalar)' LANGUAGE C;

-- Casts
CREATE implicit CAST (lvarchar AS vector WITH vector_input);
CREATE explicit CAST (vector AS lvarchar WITH vector_output);
CREATE implicit CAST (SENDRECV AS vector WITH vector_recv);
CREATE explicit CAST (vector AS SENDRECV WITH vector_send);
```

13 Deployment — From Source to Production

This section describes how to compile `vector.c` and `hnsw.c` directly on the Informix server, deploy the shared objects under `$INFORMIXDIR/extend/vector/`, adapt the SQL installation scripts, and verify the installation with a known-result SQL test — without any Java test harness.

13.1. Prerequisites

- **Informix 14.10+** installed and running; `$INFORMIXDIR` set in the environment.
- **GCC 7+** with AVX2 support on the server host (run `gcc --version` and `grep avx2 /proc/cpuinfo` to confirm).
- The DataBlade public C headers: `$INFORMIXDIR/incl/public/` (present in any standard Informix installation).
- A target database created on a dbspace with **page size $\geq$ 16 KB** (required for MAXLEN=12288). Verify with: `SELECT name, pagesize FROM sysdbspaces`. For the legacy 2,046-dim build (MAXLEN=8184), 8 KB pages suffice.
- `vector.c` and `hnsw.c` source files, plus the corresponding `_install.sql` and `_uninstall.sql` files.

13.2. Step 1 — Create the installation directory

Choose an installation directory for the `.so` files. The convention used throughout this document is `$INFORMIXDIR/extend/vector/`, which keeps the shared objects under the Informix tree alongside other DataBlade extensions. The directory must be readable by the Informix server process (`informix` user).

```
# Run as the informix user (or root, then chown)
mkdir -p $INFORMIXDIR/extend/vector
chmod 755 $INFORMIXDIR/extend/vector
```

13.3. Step 2 — Compile vector.c

The compile command mirrors exactly what the Java test harness executes internally via `sys_compile_c`. The key flags are:

- `-I$INFORMIXDIR/incl/public` — DataBlade API headers (`milib.h`, `mi.h`, etc.).
- `-O3` — maximum optimisation level; enables loop vectorisation and inlining.
- `-march=native` — generate code for the current CPU microarchitecture; unlocks AVX2 FMA instructions on modern Intel/AMD processors. **Do not use on heterogeneous clusters** where the compiled object may execute on a CPU without AVX2.
- `-fPIC -shared` — produce a position-independent shared object loadable by Informix.
- `-lm` — link the standard math library (`sqrt`, `fabs`).

```
gcc -x c \
    -I${INFORMIXDIR}/incl/public \
    -O3 -march=native \
    -fPIC -shared \
    vector.c \
    -lm \
    -o ${INFORMIXDIR}/extend/vector/vector.so
```

To build without CPU-specific instructions (portable binary, no AVX2): omit `-march=native`. The scalar fallback paths in the code are always compiled in; performance degrades roughly 2–3× for distance-intensive queries relative to the AVX2 build.

To change the maximum vector dimension, add `-DVECTOR_MAXLEN=<bytes>` before compiling. The default is already 12,288 (3,072 dims) for OpenAI `text-embedding-3-large`. To use a different size (e.g. 6,144 bytes for 1,536-dim models): `-DVECTOR_MAXLEN=6144`. Always update the `MAXLEN` value in `vector_install.sql` to match before registering the type.

13.4. Step 3 — Compile hnsw.c (optional — HNSW index support)

Only required if you intend to create HNSW approximate nearest-neighbour indexes. The `vector` type and its distance functions work without `hnsw.so`.

```
gcc -x c \
    -I${INFORMIXDIR}/incl/public \
    -O3 -march=native \
    -fPIC -shared \
    hnsw.c \
    -lm \
    -o ${INFORMIXDIR}/extend/vector/hnsw.so
```

To switch the HNSW persistence backend to filesystem storage (no BLOB table), add `-DHNSW_STORAGE_DB=0`. The default (`HNSW_STORAGE_DB=1`) stores graph data in the `hnsw_index_storage` table, which survives backup/restore cycles and requires no filesystem configuration.

13.5. Step 4 — Adapt the SQL installation scripts

Both `vector_install.sql` and `hnsw_install.sql` use a `%s` placeholder where the Java test harness substitutes the actual `.so` directory path at runtime. For manual installation, replace every occurrence of `%s` with the chosen installation path before loading the script in `dbaccess`.

```
# Replace the %s placeholder with the actual .so directory
SO_DIR="${INFORMIXDIR}/extend/vector"

sed "s|%s|${SO_DIR}|g" vector_install.sql   > vector_install_ready.sql
sed "s|%s|${SO_DIR}|g" hnsw_install.sql     > hnsw_install_ready.sql
```

After substitution, the `EXTERNAL NAME` clauses in the SQL will resolve to absolute paths such as:

```
EXTERNAL NAME '/opt/IBM/informix/extend/vector/vector.so(vector_input)'
```

13.6. Step 5 — Register the type and functions

Connect to the target database with `dbaccess` as a DBA user and run the installation scripts. The uninstall script is always safe to run first — it silently skips objects that do not exist.

```
-- 1. Clean up any previous installation
-- errors are may throw when referring to non exising vector type
dbaccess mydb vector_uninstall.sql;

-- 2. Register the vector opaque type and all SQL functions
dbaccess mydb vector_install_ready.sql;

-- 3. (Optional) Register the HNSW access method
dbaccess mydb hnsw_install_ready.sql;
```

> Informix's `LOAD FROM … DELIMITER ';'` splits the file on semicolons and executes each statement in sequence.

13.7. Step 6 — Verify the installation

Run the following self-contained SQL test. It creates a three-row table of known 3-D vectors, computes all distance metrics against a fixed query vector `Q = [2.0, 2.0, 2.0]`, and produces result sets with pre-computed expected values. The expected results are taken from `Test_XML_Select_Functions_Informix_Type_Vector` (verified against the database engine).

13.7.1. Setup — create and populate the test table

```
-- Create the test table
CREATE TABLE vec_test (id SERIAL, name CHAR(10), c VECTOR);

-- Insert three known vectors
INSERT INTO vec_test VALUES (1, 'Item A', '[1.0, 2.0, 3.0]');
INSERT INTO vec_test VALUES (2, 'Item B', '[4.0, 5.0, 6.0]');
INSERT INTO vec_test VALUES (3, 'Item C', '[7.0, 8.0, 9.0]');
```

13.7.2. Test 1 — Basic storage and text representation

```
SELECT id, name, c::lvarchar AS vec_text FROM vec_test;
```

Expected output:

```
#########################################################
#id        #name      #vec_text                        #
#########################################################
#         1#Item A    #[1.000000, 2.000000, 3.000000] #
#         2#Item B    #[4.000000, 5.000000, 6.000000] #
#         3#Item C    #[7.000000, 8.000000, 9.000000] #
```

```
#########################################################
```

13.7.3. Test 2 — Cosine distance, ordered nearest-first

```
-- Query vector Q = [2, 2, 2]
-- cosine_distance = 1 - dot(v,Q) / (|v| * |Q|)
-- Expected order: id=3 (0.0052) < id=2 (0.0131) < id=1 (0.0742)
-- id=3 is nearest because [7,8,9] points in the most similar direction to [2,2,2]

SELECT id, c::lvarchar AS vec_text,
       cosine_distance(c, vector('[2.0, 2.0, 2.0]')) AS cosine_dist
  FROM vec_test
 ORDER BY cosine_dist ASC;
```

Expected output:

```
###############################################################################
#id        #vec_text                                #cosine_dist             #
###############################################################################
#         3#[7.000000, 8.000000, 9.000000]          #                  0.0052#
#         2#[4.000000, 5.000000, 6.000000]          #                  0.0131#
#         1#[1.000000, 2.000000, 3.000000]          #                  0.0742#
###############################################################################
```

> Cosine distance measures the **angle** between vectors — it is independent of magnitude. All three vectors point in the same general direction as `[2,2,2]`; `[7,8,9]` forms the smallest angle and therefore has the smallest cosine distance (0.0052 ≈ 0°).

13.7.4. Test 3 — L2 (Euclidean) distance, ordered nearest-first

```
-- l2_distance = sqrt( sum( (v[i] - Q[i])^2 ) )
-- V1 is closest because it is geometrically nearest to Q=[2,2,2]:
--   |[1,2,3] - [2,2,2]| = sqrt(1+0+1) = sqrt(2) # 1.4142

SELECT id, c::lvarchar AS vec_text,
       l2_distance(c, vector('[2.0, 2.0, 2.0]')) AS l2_dist
  FROM vec_test
 ORDER BY l2_dist ASC;
```

Expected output:

```
###############################################################################
#id        #vec_text                                #l2_dist                 #
###############################################################################
#         1#[1.000000, 2.000000, 3.000000]          #                   1.4142#
#         2#[4.000000, 5.000000, 6.000000]          #                   5.3852#
#         3#[7.000000, 8.000000, 9.000000]          #                  10.4881#
###############################################################################
```

13.7.5. Test 4 — Inner product, L1 distance, and L2 norm

```
-- inner_product([1,2,3], [2,2,2]) = 1*2 + 2*2 + 3*2 = 12
-- inner_product([4,5,6], [2,2,2]) = 4*2 + 5*2 + 6*2 = 30
-- inner_product([7,8,9], [2,2,2]) = 7*2 + 8*2 + 9*2 = 48

SELECT id,
       inner_product(c, vector('[2.0, 2.0, 2.0]'))  AS inner_prod,
       l1_distance(c,   vector('[2.0, 2.0, 2.0]'))  AS l1_dist,
       vector_norm(c)                                AS norm
  FROM vec_test;
```

Expected output:

```
#############################################################################################
#id        #inner_prod             #l1_dist                #norm                    #
#############################################################################################
#         1#                   12.0#                   2.0#                  3.7417#
#         2#                   30.0#                   9.0#                  8.7749#
#         3#                   48.0#                  18.0#                 13.9284#
#############################################################################################
```

The L2 norm confirms |[1,2,3]| = $\sqrt{14} \approx 3.7417$ and inner_product(v, v) = norm(v)² — both are identities verified by the parity test suite.

13.7.6. Test 5 — Dimension limit

```
-- Verify the compile-time maximum and the type reject over-limit vectors

SELECT  vector_max_dims();                  --  must  return  3072  for  default  build
 (VECTOR_PAGE_SIZE_OPENAI)
SELECT vector_dims('[1.0, 2.0, 3.0]');  -- must return 3

-- Equality search
SELECT id, name FROM vec_test WHERE c = vector('[1.0, 2.0, 3.0]');  -- must return
 id=1 only
```

Expected output:

```
############################
#(expression)            #
############################
#                    3072#   # vector_max_dims()
############################

############################
#(expression)            #
############################
#                       3#   # vector_dims('[1.0, 2.0, 3.0]')
############################

########################
#id        #name      #
########################
#         1#Item A    #   # equality match
```

```
#######################
```

13.7.7. Test 6 — HNSW index (optional)

```
-- Requires hnsw.so to be installed (Step 3 above)

CREATE INDEX vec_test_hnsw_idx ON vec_test(c)
    USING hnsw_am(m='8', ef_construction='40', ef_search='20');

-- ANN search via HNSW index — must return id=3 (cosine nearest to [2,2,2])
SELECT FIRST 1 id, cosine_distance(c, vector('[2.0, 2.0, 2.0]')) AS dist
  FROM vec_test
 WHERE vector_near(c, vector('[2.0, 2.0, 2.0]'))
 ORDER BY dist ASC;
```

Expected output:

```
####################################
#id        #dist                   #
####################################
#         3#                 0.0052#
####################################
```

13.7.8. Teardown

```
DROP TABLE IF EXISTS vec_test;
```

13.8. Uninstalling

To remove all vector-related SQL objects from a database, run the uninstall scripts in reverse dependency order: HNSW first (it depends on the vector type), then vector.

> The uninstall scripts will drop the `hnsw_index_storage` table (losing all persisted HNSW indexes) and the `vector` opaque type. The type drop will **fail** if any table column still uses the `vector` type. Drop or alter all such columns before running `vector_uninstall.sql`.

```
-- 1. Drop HNSW access method and all its functions (if installed)
dbacess mydb hnsw_uninstall.sql;

-- 2. Drop vector type, all distance / utility functions, and casts
dbacess mydb vector_uninstall.sql;
```

13.9. Quick-reference — compiler flags

Compiler flag reference

Flag	Purpose	When to change
`-O3`	Maximum GCC optimisation: loop unrolling, inlining, vectorisation	Use `-O2` for faster compile during development; use `-O3` in production
`-march=native`	Enable all instructions available on the compile host (AVX2 FMA on modern x86_64)	Replace with `-mavx2 -mfma` for a portable AVX2 binary; omit for a purely scalar build
`-DVECTOR_MAXLEN=<bytes>`	Override the maximum vector byte length (default: 12288 = 3072 dims)	Default is already 12288 for OpenAI `text-embedding-3-large`. Use `6144` for 1536-dim models or `16384` for 4096-dim models; update `MAXLEN` in `vector_install.sql` to match
`-DHNSW_STORAGE_DB=0`	Switch HNSW persistence to filesystem (`$INFORMIXDIR/extend/hnsw/`)	Use when database BLOB overhead is undesirable (development, ephemeral workloads)