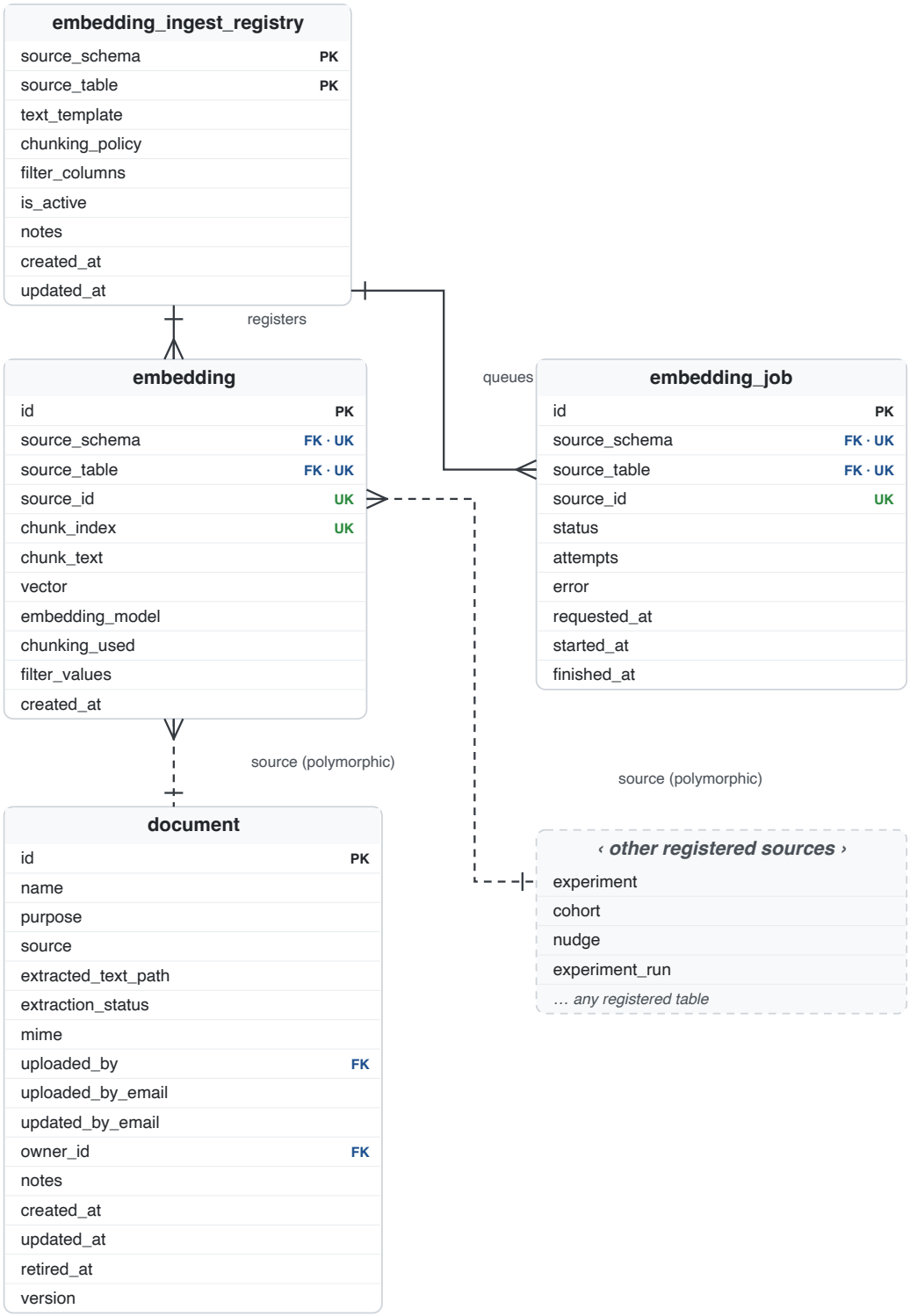
ChatMaestro — Data Model: `search`

Revision: 2026-09-20

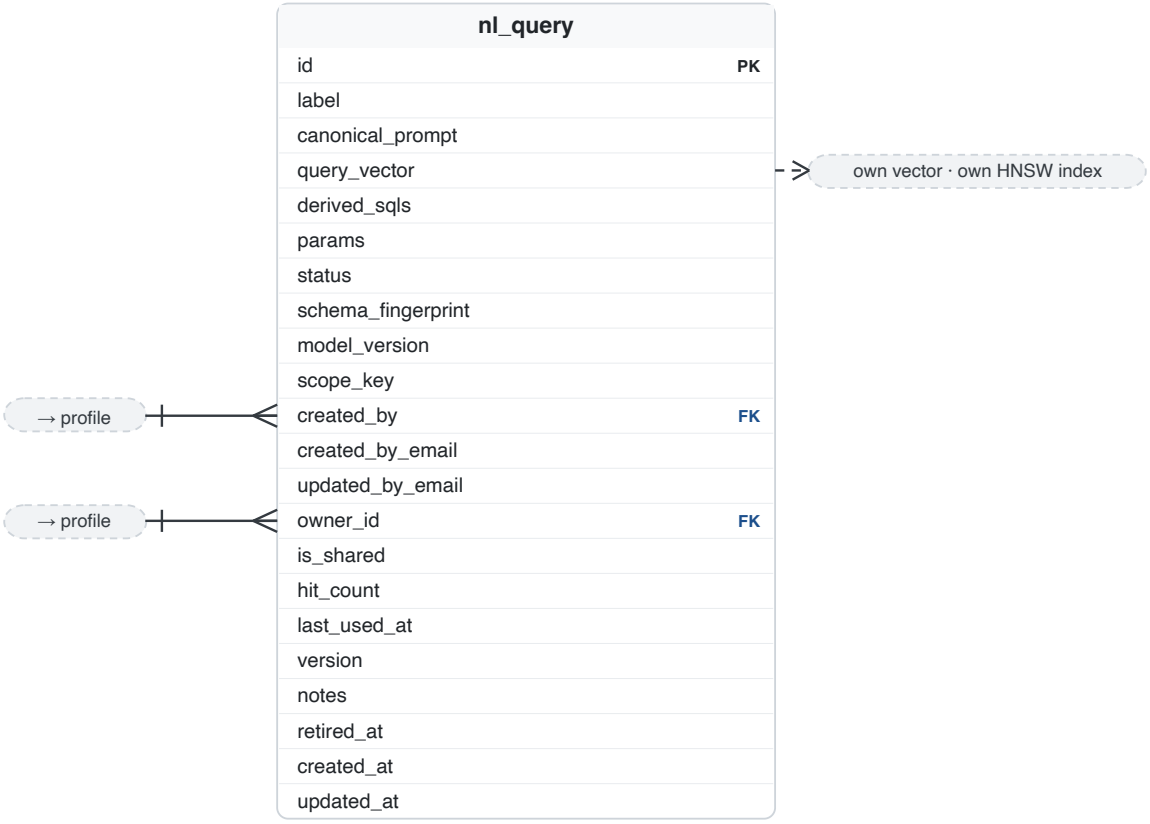
This module comprises the semantic-search sidecar — the documents, their embeddings, and the queue and registry that ingest them — and the natural-language Ask engine on the second diagram.

Entity-Relationship Diagram



The Ask Engine

A saved question, the parameterized read-only SQL queries it maps to (sometimes none — a plain-English answer), and a vector used to match a newly typed question to it. The vector is matched on this table's own index, separate from the content index above.



Narrative

What This Module Does

This module provides two things, one per diagram.

(1) Semantic search over content (first diagram). A user searches the system's written material by meaning rather than by exact words. That material is the uploaded documents and the selected descriptive text elsewhere in the app. Any table can be registered as a source. The app renders each of its rows to text and splits the text into chunks. It converts each chunk into a numeric fingerprint, called a vector, and stores every vector in one central `embedding` table that it searches by similarity. The source tables themselves are never altered, so no vector columns are added to them. The link from a stored vector back to its source row is kept by value rather than by a database foreign key, so a source can even live in another database.

(2) Asking questions in plain English (second diagram). On the Ask page a user types a question in ordinary language and gets back a short narrative answer together with zero or more result tables. Each result table can be shown as a table or a chart. Some questions need no database query at all and are answered by the narrative alone. The app keeps a set of saved questions, and each one maps to the read-only SQL queries, if any, that answer it. A newly typed question is matched by meaning to those saved questions so a good one can be reused. This lives in a single table, `nl_query`.

Reading the second diagram. It shows one table. Its `owner_id` points to the user who saved the question. That pointer is a foreign key, drawn as a solid line to a dashed off-page tag. Its `query_vector` is matched on an index kept on this table itself, shown by the own vector · own HNSW index tag. HNSW is the type of index used to search vectors quickly. This index is separate from the shared content index of the first diagram.

The Tables and Every Significant Column

embedding_ingest_registry — Deploy-Time "What/How to Embed"

One row per registered source. Its natural key is the source's name, so there is no surrogate id.

Column	Meaning · values · when populated
`source_schema`	The logical namespace of the source. For ChatMaestro's own tables it is the app schema (`chat_maestro` by default, resolved as `current_schema()`), since every ChatMaestro table lives in that one schema; for an external source it is that source's namespace, such as `remote_crm`. It is half of the composite **PK**.
`source_table`	The table to index. With `source_schema` it forms the natural composite **PK** and is what `embedding` / `embedding_job` reference.
`text_template`	A format string, set at deploy time, that turns a source row into the text to embed. It uses `{{column}}` placeholders, for example `{{title}}\n{{body}}`.
`chunking_policy`	How the rendered text is split into chunks. The value is either `none`, where the whole text is one chunk, or `fixed(size, overlap)`, a sliding window such as `fixed(800, 80)`. It is set at deploy time, chosen from the source's typical text length and the embedding model's context limit.
`filter_columns`	A jsonb list of column names to copy into `embedding.filter_values` for scoped search. For the document source this includes `purpose`, so a semantic pull can restrict to `context` or `reference` chunks.
`is_active`	The switch that pauses a source. It turns embedding on or off for this source without deleting its configuration. A disabled source keeps its registry row and any existing vectors, and it stops queuing new work to embed or re-embed rows. The default is `true`.
`notes`	Free-text notes a user writes about this registered source and its embedding configuration. The notes are searchable by SQL keyword. The sidecar's own configuration is not itself embedded.
`created_at / updated_at`	When the config row was created / last edited (`updated_at` bumps on any config change).

embedding — The Polymorphic Vector Sidecar (One Row per Chunk)

Column	Meaning · values · when populated
`id`	The surrogate **PK** for the chunk row.

Column	Meaning · values · when populated
`source_schema, source_table`	Together with `source_id`, these form the virtual pointer to the source row. The pair `(source_schema, source_table)` is a composite **FK** to the registry. Both columns are also part of the unique key below, which is why they are marked **FK·UK**.
`source_id`	The primary-key value of the source row. It carries no database foreign key, because it can point at rows in different tables. It is part of the unique key, which is why it is marked **UK**.
`chunk_index`	The chunk number within the source row, counting from zero. This is what lets one source row map to many `embedding` rows. The full natural key is unique across `(source_schema, source_table, source_id, chunk_index)`, so every one of those columns is marked **UK**.
`chunk_text`	A stored copy of the source text slice this vector was built from. It is a deliberate duplicate. The app keeps it so it can show a search result and re-embed the text later without fetching the source again.
`vector`	The embedding itself, of type `vector(1024)`. Every install uses this one fixed dimension, and larger models are truncated to 1024. The distance metric, cosine, is fixed on the HNSW index rather than stored here.
`embedding_model`	Which model produced this vector, named with its provider, for example `openai/text-embedding-3-large`. Vectors from different embedding models are not comparable, and every vector shares one HNSW index. Without this column, changing the install's embedding model would silently mix incomparable vectors, and similarity search would degrade with nothing in the data to reveal it. The column lets a query filter to a single model, and it lets a re-embed find exactly the rows built by a superseded model. Provider choice is therefore limited to models that are 1024-dimension or can be truncated to 1024.
`chunking_used`	The chunking policy that produced this chunk, recorded for the same reason as `embedding_model` beside it: `chunking_policy` on the registry is the setting, and this is what actually happened. Changing the policy invalidates every chunk cut under the old one, because a query then compares text split one way against text split another. Without this column that change is silent, since nothing else records how an existing chunk was made. With it, the rows still to convert are exactly those whose policy differs from the one now in force, so a policy change reuses the model-change machinery in `ALGORITHMS §9` rather than needing its own. A rebuild is not atomic, so the index legitimately holds chunks under both policies while it runs, and this is what keeps that state legible.
`filter_values`	A jsonb map of the filter columns copied for this row, for example `{"category":"billing","language":"en"}`. It enables `WHERE`-style filtering alongside the vector match. It is populated at embed time from the source row, following `registry.filter_columns`.
`created_at`	When the chunk was embedded. It is compared against the source's `updated_at` to detect staleness.

embedding_job — The Async (Re)Embed Queue

A short-lived work queue rather than a history log. When everything is fresh, it is close to empty. It holds only outstanding or in-flight work, with at most one open job per source row.

Column	Meaning · values · when populated
`id`	The surrogate primary key for the row, marked **PK**.
`source_schema, source_table, source_id`	Which source row to embed or re-embed. The pair `(source_schema, source_table)` is the composite **FK** to the registry. The three columns together carry the partial-unique rule of one open job per source row, which is why they are marked **UK**.
`status`	The job's state. It moves from `pending` to `running`, and then to either `done` or `failed`.
`attempts`	The number of attempts so far, used to decide backoff and when to give up.
`error`	The detail of the last failure, set when `status = failed`.
`requested_at / started_at / finished_at`	Lifecycle timestamps that record when the job was queued, picked up, and completed.

document — The File-or-URL Store

Column	Meaning · values · when populated
`id`	The surrogate **PK**. It is also the `source_id` when a document is embedded.

Column	Meaning · values · when populated
`name`	The document's display name or filename.
`purpose`	The document's category, which scopes retrieval: • `context` — grounds the enrollee chat through retrieval-augmented generation (RAG), the technique of feeding retrieved text to the model as background. It is attached to an experiment through `experiment_context_file`, shared by all of that experiment's runs, and it is owned by the experimenter who uploaded it. • `reference` — admin-owned methodology material that the Ask engine pulls in by meaning. The set of categories is extensible, and a new category is a new retrieval scope.
`source`	The document's origin, held in one field. It is either an `http(s)://` URL for a web-hosted document fetched over the Internet, or, when there is no `http(s)://` prefix, the object-storage key of an uploaded file the system holds. The default object store is Cloudflare R2, and the provider is set in `.env`. One field means a user never has to fill two, and the app branches on the URL scheme.
`extracted_text_path`	The stable object-storage key of the cached extracted-text snapshot, derived once from `id`, for example `extracted/<id>.txt`. The middle tier extracts text from the file, covering Word, Excel, and PDF, including scanned PDFs read by optical character recognition (OCR). Embeddings and RAG retrieval are built from this snapshot. The key is never cleared or repointed: a re-extraction writes to a staging key and atomically overwrites this one on success, so the live snapshot never goes offline. It is kept human-locatable for diagnostics.
`extraction_status`	The state of the `extracted_text_path` file, and whether there is anything to read at all. The snapshot stays readable in every state except `pending`. A re-extraction is written to a staging key and swapped in atomically, so retrieval keeps serving the existing text for the whole time a new extraction runs, and keeps serving it even if that extraction fails. The states are: • `pending` — never extracted, with nothing to read yet. This and a `failed` whose first extraction never succeeded are the two unreadable states; every other state serves text. • `ready` — holding the current, complete extraction. • `stale` — holding a complete but outdated snapshot because the source changed and a re-extraction is queued or running, while retrieval keeps using this text until the swap. • `failed` — the last attempt errored, and any previous snapshot is still served. A `stale` file never returns to `pending`, so a document that has once extracted is never empty again.
`mime`	The content type of the file, which guides text extraction at ingest.
`uploaded_by`	A foreign key to `profile` naming the uploader. It is kept for provenance and never changed, and it is nullable for system uploads.
`owner_id`	A foreign key to `profile` naming the current owner. For a `context` document it defaults to the uploader. That experimenter re-uploads, retires, and decommissions it, while peers reference it as run context, and the admin can transfer ownership. A `reference` document is admin-managed: the admin creates, reads, updates, and deletes it, experimenters read and use it, and there is no transfer among experimenters, so its owner stays the admin. The value goes null if the owner is deleted, which leaves the document admin-managed.
`notes`	Free-text notes a user writes about this document. They are searchable by SQL keyword, and in the semantic index by meaning.
`created_at / updated_at`	A change to `updated_at` drives embedding staleness, so a changed document queues a re-embed.
`retired_at`	A soft-retire marker. A non-null value hides the document from active experiment composition while keeping it for the experiments that already reference it as context. Decommissioning is a hard delete, allowed only when no experiment references the document. So `retired_at` is what removes a still-referenced document from the active pool.
`version`	A counter that supports optimistic concurrency control (OCC), which stops two users from overwriting each other's edits, for the editable metadata: `name`, `source`, owner, and `retired_at`. A stale save is rejected with the message "record changed — reload." The extracted-text snapshot has its own concurrency control through `extraction_status`, using the staging-and-swap flow rather than this counter.

The two purposes scope retrieval in two ways. A `context` document is attached to an experiment through `experiment_context_file`, and enrollee RAG draws only on that experiment's attached documents. A `reference` document has no per-run link. The Ask engine reaches it by a semantic pull scoped to `purpose = reference`, a top-K vector match — the closest few matches — that can span several reference documents, so an answer can be grounded in methodology material. Reference documents are admin-owned and admin-managed, like the model catalog, and experimenters read and use them.

How They Relate (Reading the Lines)

- registers: one registry row governs many `embedding` rows, through a composite **FK** on `source_schema, source_table`.
- queues: a registered source queues many `embedding_job` rows.
- source (polymorphic), the dashed line: each embedding points at one source row, and one source row maps to zero or more embeddings, one per chunk, which is what `chunk_index` counts. The relationship is zero-to-many rather than one-to-one, because a not-yet-embedded document has zero rows. The tie is virtual, with no database foreign key, which is what lets a source live in another schema or a remote system.

Config and Behavior (Not Columns)

The active embedding model and distance metric, cosine, are a single install setting in `.env` rather than a table. Freshness is decided by comparing `embedding.created_at` against the source's `updated_at`. Local sources queue jobs through AFTER INSERT and AFTER UPDATE triggers, and an AFTER DELETE trigger clears their rows. Remote sources queue jobs through the app or connector, or through a change feed.

Principle of Operation

Three notions have similar names and are easy to conflate:

Term	Lives in	Is a…	Meaning
`text_template`	registry (per source)	format string	Which columns' text becomes the vector, and how it is laid out. This is the content that semantic search matches on.
`filter_columns`	registry (per source)	list of column names	Which columns to copy as structured filters. This is configuration, written once per source.
`filter_values`	embedding (per chunk)	key→value map	The actual values of those columns for one chunk's source row. This is data, stamped on every embedding row.

In short, `filter_columns` is the recipe, such as "copy category, language…", and `filter_values` is the cooked result, such as "category=billing, language=en", stamped onto each chunk. The values are copied, or denormalized, so a similarity search never has to join back to the source. That matters because the source may be in another schema or a remote database.

Registered Sources in This Deployment

The mechanism above is configured for ten content sources, the tables whose `notes` or free text is embedded. Each row is one `embedding_ingest_registry` record. Its `text_template` is the text that becomes the vector, and its `filter_columns` are the structured values copied onto every chunk so a search can be narrowed, for example to one experiment or one cohort — a named group of enrollees. Keyword search still reaches every column in the database. The sources below are the ones additionally searchable by meaning.

source	`text_template` — vectorized text	`filter_columns` — narrowing values
`chat_maestro.document`	name · extracted text · notes	purpose, mime, uploaded_by
`chat_maestro.experiment`	name · description · scenario · system_prompt · opening_message · notes	id, created_by
`chat_maestro.experiment_run`	notes	experiment_id, nudge_id, cohort_id, model_config_id, state
`chat_maestro.cohort`	notes	created_by
`chat_maestro.nudge`	name · text · notes	owner_id, created_by
`chat_maestro.email_template`	notes	kind
`chat_maestro.model_config`	notes	combine_method, created_by

source	text_template — vectorized text	filter_columns — narrowing values
chat_maestro.model_catalog	notes	provider, enabled
chat_maestro.nl_query	notes	status, scope_key

chat_maestro here is this install's app schema, which is configurable (all ChatMaestro tables live in it; the modules are logical groupings, not separate schemas). An external source would instead show its own namespace, such as remote_crm.

The Ask engine's saved questions, in nl_query, sit in two places for two jobs. Their notes are embedded into the shared content index like any source above, so a content search can surface them. Matching a newly typed question to a saved one uses a separate vector, query_vector, on the nl_query table's own index. So searching your material and matching a saved question stay two separate lookups.

Some tables are not embedded, on purpose, for four reasons:

- Privacy — profile notes, which are personal data (PII) about users, and audit_log, the forensic trail, stay searchable by SQL only, so the shared index holds little personal data.
- Scale and blinding — the raw transcript, in chat_round and chat_round_candidate, and the out-of-band message channel are kept out. Embedding every turn would swamp the index, and both are reached through the natural-language-to-SQL side instead.
- No free text worth embedding — the pure join, queue, numeric, and sink tables (cohort_member, run_enrollment, experiment_context_file, message_recipient, run_result, and model_config_model).
- Mechanism, not content — the sidecar's own plumbing (embedding, embedding_job, and embedding_ingest_registry) is not a source of itself.

Searchable is not the same as embedded. Every text column, and every notes field anywhere in the schema, is reachable by keyword and full-text SQL and by the natural-language-to-SQL side of the Ask; keyword search over the free-text columns is backed by a per-table GIN full-text index (see "Indexing and access paths" in the overview). Embedding a column additionally places it in the semantic vector index, so it can be matched by meaning. The two coexist, and a short note is found either way. Every notes column is embedded except the two privacy carve-outs above, which keep their keyword index but not the semantic one. Some notes run short or null, so their semantic recall is modest. That is harmless, because keyword search still covers them and the shared index is small at this scale.

Understanding Vector Embedding Concepts

This section explains how embedding works, using small illustrative examples. Some of them use a made-up source table, kb.article, purely to show the mechanics. That table and its columns are examples only — they are not part of the ChatMaestro schema.

What text_template and chunking_policy Look Like

text_template picks which columns to embed and joins them into one small block of text. The separators between fields, usually newlines, and any field labels are not decoration; they matter for two plain reasons. First, a separator keeps the end of one field from running into the start of the next as a nonsense word — "Refund policy" followed by "When a refund…" must not become "Refund policyWhen a refund…". Second, a label lets the embedding model tell the fields apart, and sometimes the label itself carries meaning: if chat transcripts were ever embedded, marking each line user: or assistant: would change the meaning, since who said a sentence matters — though ChatMaestro does not embed transcripts, so that stays a what-if. For ChatMaestro's own sources the fields are simply joined with newlines, which is enough. Examples across sources:

```
chat_maestro.document   -> {{name}}\n{{extracted_text}}\n{{notes}}
chat_maestro.experiment ->
{{name}}\n{{description}}\n{{scenario}}\n{{system_prompt}}\n{{opening_message}}\n{{notes}}
kb.article (external)   -> {{title}}\n{{summary}}\n\n{{body}}
```

chunking_policy is either none (one chunk) or fixed(size, overlap) — a sliding window, e.g. fixed(800, 80) is ~800-token windows overlapping by 80 (the overlap keeps an idea that straddles a boundary intact). fixed(size) over text shorter than size yields one chunk anyway, so it degrades gracefully to none.

Worked Example: A Source Table Kb.Article

A generic illustrative source — the sidecar is schema-agnostic. It has multiple embed columns, some filter columns and some non-filter columns, and columns used for nothing:

Column	Type	Role
`id`	uuid PK	→ becomes `source_id`; not embedded, not a filter
`title`	text	embed
`summary`	text	embed
`body`	text (long)	embed
`category`	text	filter
`product_area`	text	filter
`language`	text	filter
`is_published`	boolean	filter
`author_email`	text	exclude (not embedded, not a filter)
`view_count`	int	exclude
`internal_notes`	text	exclude — internal; must not leak into retrieval
`updated_at`	timestamptz	exclude as a field, but drives staleness

Two representative rows:

```
id="a1…"  title="How refunds are processed"  summary="Refund timing, eligibility."
          body="<~1500 chars…>"  category="billing"  product_area="invoicing"
          language="en"  is_published=true  author_email="jo@acme.com"  view_count=4210
          internal_notes="TODO: revise for 2026"  updated_at=2026-06-01

id="b2…"  title="Resetting your password"  summary="Reset via the email link."
          body="<~300 chars…>"  category="account"  product_area="auth"
          language="en"  is_published=true  …
```

The registry row that configures this source:

```
source_schema  = "kb"
source_table   = "article"
text_template  = "{{title}}\n{{summary}}\n\n{{body}}"
chunking_policy = "fixed(800, 80)"
filter_columns = ["category", "product_area", "language", "is_published"]
is_active      = true
```

`internal_notes`, `author_email`, `view_count` appear in neither `text_template` nor `filter_columns` — so they are simply *excluded* (never touch the vector, never become filters). "Exclude" is the residual, not a stored list.

What the pipeline emits — article `a1` has a long body, so `fixed(800,80)` splits it into 2 chunks; the same `filter_values` are stamped on both:

```
(source_schema="kb", source_table="article", source_id="a1…", chunk_index=0,
 chunk_text="How refunds are processed\nRefund timing, eligibility.\n\nWhen a refund…(~800)…",
 vector=[0.0123, -0.044, … 1024 floats …],
 filter_values={"category":"billing","product_area":"invoicing","language":"en","is_published":true})

(… source_id="a1…", chunk_index=1,
 chunk_text="…(overlapping next ~800 chars of body)…",
 vector=[…1024…],  filter_values={ …same as above… })

(… source_id="b2…", chunk_index=0,   # short body -> a single chunk
 chunk_text="Resetting your password\nReset via the email link.\n\nClick 'Forgot password'…",
 vector=[…1024…],  filter_values={"category":"account","product_area":"auth","language":"en","is_published":true})
```

A hybrid query — "how do I get my money back", within `billing`, published only — is a single-table scan, no join back to `kb.article` (which may be remote):

```
SELECT source_id, chunk_text
FROM embedding
WHERE source_schema='kb' AND source_table='article'
  AND filter_values @> '{"category":"billing","is_published":true}'  -- GIN(jsonb_path_ops) on filter_values
ORDER BY vector <=> :query_vector      -- cosine distance (fuzzy match), HNSW on vector
LIMIT 5;
```

The scope filter uses jsonb *containment* (@>) so it is served by a GIN index on `filter_values`, and the similarity ordering is served by the HNSW index on `vector` — no join back to `kb.article`, which may be remote.

Why This Shape

- Denormalized `filter_values` means filtering needs no join to the source. The vector search is one indexed table scan, and it still works when the source is remote.
- `filter_values` is jsonb rather than typed columns, because `embedding` is polymorphic and the keys differ per source: a `kb.article` chunk has `category`, and a `chat_round` chunk has `run_id`.
- The virtual `(source_schema, source_table, source_id)` key has no database foreign key, so any source can be embedded, local or remote, without altering it.
- The template matters, because embeddings are over text, and how a row is serialized changes what the vector captures.

The Ask Engine — Asking Questions in Plain English

The Ask page lets a user get answers by typing a question the way they would say it — “how did the runs for a given model do?” — and the engine writes the database query for them. An answer comes back as a short narrative description together with zero or more result tables, and each result table can be shown as a plain table or as a chart. Some questions need no database query and are answered by the narrative alone. The app keeps a growing set of saved questions; each records the plain-English wording, the read-only SQL queries (if any) it maps to, and a numeric fingerprint of its wording used to recognize the same question again. All of this lives in one table, `nl_query`.

How It Works, from the User's Point of View

1. You type a question. The app compares its meaning against the saved questions you are allowed to see — your own, ones a teammate has shared, and the official ones.
2. If a saved question matches, the app reuses *its* query. If your question mentions a specific value — a model name, a date — that value is slipped into the saved query in place of a blank, and the results appear. Nothing new is saved.
3. If nothing matches, the app works out how to answer your question on the spot — running a read-only query when one is needed — and shows a short written answer with any result tables (each as a table or a chart). Some questions are answered in words alone. This one-off is not kept.
4. If you want to keep it, you click Save prompt. You give it a short button name, and for any specific values in your question you choose whether to turn each into a fill-in blank (so the button asks for that value next time) or keep it fixed (so “Top 5 models” always means five, with nothing to fill in). Only then does it become a saved question — yours, and private to you until you choose to share it. You are never asked to keep the questions you don't care about, so the list stays clean.

One Saved Question, Many Values

A saved question can contain blanks — for example “results for model ___”. The query behind it is written once with those blanks left open; each time it runs, the specific values are filled in and the query is executed. So asking about one model today and a different model tomorrow reuses the *same* saved question — the app just fills the blank differently. A question with no blanks simply runs as it is. This works the same for questions answered in words alone: “summarize the findings for model ___” is one saved question that serves every model, so there is no need to keep a separate one per value.

Not every specific value has to become a blank. When you save a question you decide, value by value, which are fill-in blanks and which stay fixed. “The top 5 models” can keep the 5 fixed, so the button runs on one click with nothing to ask — you are not made to answer “how many?” just to get the five you meant.

Who Can See and Trust a Saved Question

Every saved question has one of three levels:

- Draft — someone saved it; it is private to them (and only they can edit it) unless they share it read-only with the team.
- Approved — an administrator has checked it, so everyone may read and run it, and only an administrator may change it.
- Built-in — it ships with the install as part of a starting set, with the same access as an approved question (administrators can edit; experimenters can read and run).

A frequently-used draft is a natural candidate for an administrator to approve. These levels are independent of how a question is answered — a question at any level may be answered in words alone or produce one or more result tables.

nl_query — Every Column

Column	Meaning
`id`	The unique identifier for the row, marked **PK**.
`label`	The short button name shown in lists and menus, for example “Model results,” given by the user when they save the question. It is kept short for the screen and is not the full question text. If it is left empty, the app shows a shortened `canonical_prompt` instead. It is unique per owner among live, labeled questions (a partial unique on `(owner_id, label)`), so one owner cannot save two questions under the same name; it is not constrained across owners. When a list mixes owners, two identical labels are told apart by their origin — the viewer’s own, shared by a named teammate, approved, or built-in — and by the full-question `canonical_prompt` subtitle.
`canonical_prompt`	The saved question in plain English, normalized so it is semantically equivalent to every way of asking it. A specific model name becomes “a given model,” and a value kept fixed stays as it is. This is the text turned into `query_vector`, so a newly typed question can be matched to it. It is also shown as the full-question subtitle or tooltip. The short button caption is `label`, not this text.
`query_vector`	The numeric fingerprint of `canonical_prompt`, written when the question is saved. It sits on an index kept on this table, so a newly typed question is matched against the saved ones in the same request.
`derived_sqls`	The read-only database statements the question maps to — none, one, or several — kept together as an ordered list. None means the question is answered by the narrative alone. Each entry also carries a display hint (a plain table, or a chart such as bar / line / pie). The statements have blanks for their parameters; the actual values are supplied when the question runs.
`params`	The list of the question's parameters — each with a name, a type, whether it is required, and a default. The one definition is used three ways: to check the values supplied, to build the small form that asks for any missing value, and to tell the language model what to fill in. Which values become parameters is the user's choice when saving: a value kept fixed is baked into the query and is *not* listed here (so “Top 5 models” needs no form). Parameters apply whether or not the question runs a database query — a words-only answer can be parameterized too.
`status`	The trust level, one of three: • `draft` — a user saved it, and it is private to its owner, who may edit it, unless it is shared read-only. • `approved` — an administrator vetted it, so everyone may read and run it, and only an administrator may edit it. • `system` — it ships with the install, with the same access as approved, so administrators can read and write it and experimenters can read it. The level is independent of how the question is answered. An administrator moves a question between `draft` and `approved` in either direction: approving promotes a draft, and unapproving hands an approved question back to its owner as a draft, taking the button off everyone else’s Ask surfaces at their next load. Each move writes an `audit_log` row.
`schema_fingerprint`	A fingerprint of the database structure the saved statements were written against. If the structure later changes, the statements are re-generated from `canonical_prompt` before use.
`model_version`	Which language model produced the saved statements; used together with `schema_fingerprint` to decide when they must be re-generated.
`scope_key`	The context a saved question belongs to — a context-type tag such as `results`, `experiments`, `cohorts`, `models`, `documents` or `global`. It decides where the question is offered (a screen’s Ask shows its own context plus the global and built-in ones) and seeds default values from that screen (such as the run you are looking at). It does not grant access — every run is still bounded by what the asker is permitted to see. It names a context *type*, not one specific run, so a question stays reusable across runs.

Column	Meaning
`created_by`	A foreign key to `profile` naming the author who first composed the question. It never changes, and it is empty for built-in questions.
`owner_id`	A foreign key to `profile` naming the current owner, which defaults to the author. The owner holds edit rights on the draft. The admin can transfer ownership. It is empty for built-in questions.
`is_shared`	The owner's switch to let teammates see and run their draft read-only, while the owner keeps the only edit rights. Approved and built-in questions are readable by everyone regardless, unless retired.
`hit_count`	How many times the saved question has been reused. It is used to surface popular questions and to suggest drafts worth approving.
`last_used_at`	When it was last run. It powers a "recently used" list.
`version`	A counter bumped on every edit, so two users editing at once cannot silently overwrite each other.
`notes`	Free-text notes, edited on the Saved-questions screen by whoever may edit the row: the owner of a draft, an administrator for anything. An administrator's approval note is appended here, stamped with who wrote it and when, so the reason a question was approved travels with the question. Findable by keyword search; not part of the matching, which uses only `canonical_prompt`.
`retired_at`	A soft-retire timestamp that withdraws an approved question without deleting it. When set, the question disappears from every Ask surface and from the match set, while the row, its SQL, its use count and its notes stay. Only an administrator sets it, and only on an approved question; clearing it restores the question, so retirement is the undoable way to withdraw a shared button. Deletion is the separate, permanent way: a draft may be deleted by its owner or an administrator, and an approved question, retired or not, by an administrator. A built-in question is never retired or deleted.
`created_at / updated_at`	When the saved question was created and last changed.

Internal Design Notes

This section is for those building and maintaining the system. It records the reasoning behind the choices above; none of it is needed to understand or use the features. A reader who only wants to know what the system does can stop at the previous section.

Why the Ask Engine Is a Single Table

An alternative design would split this into three tables: a governed catalog of query "templates," a table of saved buttons that bind some of a template's parameters, and a separate short-lived cache of freshly generated queries. Those three would share most of their columns, so the split buys little. Treating "a saved question" as one kind of thing at one of three trust levels (`draft`, `approved`, `system`) is cleaner: promoting a popular draft to an official question is a one-field change, and there is only one place to search when matching a typed question.

The Prepare / Bind / Execute Model

Each saved question behaves like a prepared statement in a traditional database application: the query text (with blanks) is written once and stored in `derived_sqls`; each run binds the specific values and executes. No row is created per run, and different parameter values do not create new saved questions — they reuse the same one. That is why a single saved question serves every value of its parameters.

Saving Is Explicit, Not Automatic

When a typed question matches nothing, the app answers it once and does not store it. It becomes a saved question only if the user clicks Save prompt. This keeps the table full of deliberately-kept questions rather than every one-off anyone ever tried, and it leaves the user in control of what is worth keeping. (A newly generated query is still looked up against the saved questions first — the lookup always happens; only the *storing* waits for an explicit save.)

Why the Ask Engine Keeps Its Own Vector

Matching a typed question to a saved one must happen *while the user waits*, in the same request. Content search (the first diagram) embeds its material in the background through a queue — fine for documents, too slow for a live keystroke-to-answer match. So the Ask engine keeps its matching vector on its own table with its own index, separate from the shared content index.