

Hybrid Search in PostgreSQL

Combining Vector and Full-Text for Real-World Applications

Shayon Sanyal, Principal Worldwide PostgreSQL Specialist SA, AWS

PGConf.EU 2026, Valencia

One title. Four different movies.

Search: **“Titanic”**

BM25



Titan A.E.
2000

Score blend



Raise the Titanic
1980

+ Box office



Titanic
1997 ($2.2B)

RRF + rerank



Titanic
1953

Which movie did the person mean? A title alone leaves the intent open.

7,531 movies from the CMU Movie Summary Corpus. Blend: bge-small + BM25. This title query has no judged answer.

The movie dataset

PART	WHAT WE USE
Documents	**7,531 movies** from the **CMU Movie Summary Corpus**
Searchable text	Title + plot + cast, shared by BM25 and each main vector model
Questions	**TREC Tip-of-the-Tongue**: people describing a forgotten movie
Search quality	Tested on **43 questions** we hadn't tuned for
Teaching examples	**Titanic** title and **Toy Story** clue, outside evaluation

Each graded question has one known movie. Follow its position through the rankings.

NDCG@10 (Normalized Discounted Cumulative Gain) measures the top 10 results, rewarding relevant answers near the top. Shown on a 0–100 scale; 100 is ideal.

Subset: movies with plot, year and box-office metadata. Sources: CMU corpus and TREC questions

This demo runs on localhost:5433

psql — movies_tot

```
movies_tot=# \conninfo
 Database                | movies_tot
 Host                    | 127.0.0.1
 Server Port             | 5433
movies_tot=# SHOW server_version;
 server_version
----------------
 18.6 (Homebrew)
(1 row)
movies_tot=# \dx
     Name       | Version
----------------+---------
 pg_textsearch  | 1.4.0
 plpgsql        | 1.0
 vector         | 0.8.6
(3 rows)
```

This laptop

`127.0.0.1:5433`

Database: `movies_tot`.

PostgreSQL 18.6

Local Homebrew installation.

Two search extensions

pg_textsearch 1.4.0: BM25

vector 0.8.6: pgvector + HNSW

Text, terms, and vectors in one row

```sql
CREATE TABLE docs (
  id              text PRIMARY KEY,
  body            text NOT NULL,
  tsv             tsvector GENERATED ALWAYS AS (to_tsvector('english', body)) STORED,
  embedding       vector(1536),  -- Cohere Embed v4
  embedding_local vector(384)    -- bge-small-en-v1.5
);
CREATE INDEX docs_tsv_gin ON docs USING gin (tsv);
CREATE INDEX docs_embedding_hnsw ON docs USING hnsw (embedding vector_cosine_ops);
CREATE INDEX docs_embedding_local_hnsw
  ON docs USING hnsw (embedding_local vector_cosine_ops);
CREATE INDEX docs_body_bm25 ON docs USING bm25 (body) WITH (text_config = 'english');
```

Core PostgreSQL: text search + GIN. **pgvector:** vectors + HNSW.
pg_textsearch: ranked BM25. Its index name is used in the query.

The generated tsvector follows the text. **Refresh embeddings when text changes; never mix models.**

One row: Toy Story

```
SELECT id, body, tsv, embedding, embedding_local
FROM docs WHERE id = '53085';
```

COLUMN	STORED VALUE
`id`	`53085`
`body` Title + plot + cast	**Toy Story** Woody is a pull-string cowboy doll and leader of a group of toys … Cast: John Ratzenberger, Tim Allen, Annie Potts, …
`tsv`	`… 'cowboy':9 'doll':10 'group':15 …`
`embedding`	`[0.028936684, -0.027140614, 0.012622381, …]` 1,536 numbers from Cohere Embed v4
`embedding_local`	`[0.0006861224, 0.0110116955, 0.0039944383, …]` 384 numbers from bge-small-en-v1.5

Text and vectors shortened. `tsv` shows selected terms and their word positions.

FTS: which words must match?

Full-text search (FTS) matches normalized words. A `tsvector` stores document terms; a `tsquery` specifies the match.

Remembered words

"hero bat princess"

Require all three (AND)	**0 movies**
Allow any word (OR)	**562 movies**

Wolfhound is #47 with OR: it matches `bat` and `princess`. Requiring `hero` excludes it.

```
-- Allow any of the remembered words.
WITH query AS (
  SELECT to_tsquery('english',
    'hero | bat | princess') AS q
)
SELECT id, ts_rank_cd(tsv, q) AS rank
FROM docs, query
WHERE tsv @@ q
ORDER BY rank DESC, id
LIMIT 50;
```

`@@` matches 562 movies; `ts_rank_cd` orders them; `LIMIT` keeps the first 50.

Tradeoff: AND can drop the answer; OR can admit hundreds of movies.

On the 43 full test clues, requiring every term finds nothing for 38. Matching and ranking are separate steps.

BM25: give distinctive words more weight

BM25 weights matching words. Here, **pg_textsearch** supplies it.

Search clue

"A cowboy **doll** named **Woody** is jealous when a new **Buzz Lightyear** action figure becomes **Andy**'s **favorite** toy."



#1 Toy Story 67.23

doll + favorite: **+7.05**



#2 Toy Story 2 60.35

doll + favorite: **0**

Lightyear occurs in **3** of 7,531 movies; **new** occurs in **3,141**. Rare matches carry more weight.

```sql
SELECT id, -neg_score AS bm25
FROM (
  SELECT id,
    body <@> to_bm25query(
      (SELECT body FROM queries
       WHERE id = 'movie-toy-story'),
      'docs_body_bm25'
    ) AS neg_score
  FROM docs
  ORDER BY neg_score LIMIT 50
) hits
WHERE neg_score < 0
ORDER BY neg_score, id;
```

Use the full stored clue. `<@>` returns negative BM25: sort ascending, flip the sign for display.

BM25: reward rare terms, limit gains from repetition, and account for document length.

Orange = doll + favorite; gray = other matched words. Positive BM25 scores for the full clue, outside evaluation.

pgvector: when the wording changes

pgvector stores and searches vectors. An **embedding model** turns movie text and clues into vectors.

Remembered scene (excerpt)

“two men driving a car … then go to vegas … sees **dinosaurs**.”

Fear and Loathing in Las Vegas

BM25 #28 → vector **#1**

The clue says **dinosaurs**; the plot describes hallucinated **lizards**.

```sql
BEGIN;
SET LOCAL hnsw.ef_search = 100;
SELECT id,
  embedding <=> (
    SELECT embedding FROM queries
    WHERE id = '297'
  ) AS distance
FROM docs
WHERE embedding IS NOT NULL
ORDER BY distance
LIMIT 50;
COMMIT;
```

The vector comes from the full stored clue. `<=>` is cosine distance: smaller is closer. Keep ascending distance + LIMIT.

Pitfall: a similar scene can match even when a required word is absent.

Embed v4 (1,536 values). HNSW is an approximate vector index; this small table uses an exact scan. Check EXPLAIN.

Two embedding models, one column each

	COHERE EMBED V4	BGE-SMALL-EN-V1.5
Runs on	Amazon Bedrock	This laptop, with fastembed
Numbers per vector	1,536	384
Column	`embedding` + HNSW index	`embedding_local` + HNSW index
Used for	Meaning, RRF, rerank, evaluation, `hybrid_search()` and agent tools	Score blend, filter demo, `hybrid_search_blend()` and box-office boost

Compare a question only with vectors from the same model. Re-embed when the text or the model changes.

Reranking uses Cohere Rerank 3.5 on Amazon Bedrock. Claude Opus 5.5 only runs the agent.

Before combining: the scores have different scales

WOLFHOUND CLUE (TOP 50 CANDIDATES)	LOWEST	HIGHEST
BM25, positive for display	29.713	49.267
Embed v4 cosine similarity	0.342	0.427

```
-- A tempting mistake:
SELECT doc_id, bm25_score + cosine AS score FROM candidates;
```

Pitfall: adding raw scores lets the larger numerical scale dominate.

Two ways forward: **use ranks** with RRF, or **normalize scores** before blending.

Both lists answer the same question. These scores are not probabilities.

3. Combine ranks

Give the same movie credit from both lists

"a little damaged bat as his pet … saving a princess … It's not the Beastmaster."

$$RRF(d) = \Sigma\ 1 / (60 + rank_{list}(d))$$ Missing from a list? Add 0.

Reciprocal Rank Fusion (RRF) adds rank-based credit from each list. Each retrieves up to 50 movies; missing from that list contributes zero.

	TEXT SEARCH (BM25)	VECTOR SEARCH (EMBED V4)	COMBINED BY RRF
1	Un Argentino en New York	The Swan Princess	**Wolfhound (Volkodav)**
2	Hollywood Shuffle	Quest for Camelot	Ace Ventura: When Nature Calls
3	Journey to the Center of the Earth	3–9 Ranks omitted	3–5 Ranks omitted
4	**Wolfhound (Volkodav)**	10 Ace Ventura: When Nature Calls	6 The Swan Princess
5	Ace Ventura: When Nature Calls	11 **Wolfhound (Volkodav)**	7 Un Argentino en New York

Wolfhound: `1/(60 + 4) + 1/(60 + 11) = 0.02971 → #1`

Each single-list leader: `1/(60 + 1) + 0 = 0.01639`

Green = known answer: Volkodav from the Grey Hound Clan (2007), also titled Wolfhound.

Fuse the same BM25 and vector lists

Inputs: the clue text and its Embed v4 vector. **Output:** up to 50 candidates for reranking.

```
BEGIN; SET LOCAL hnsw.ef_search = 100;
WITH keyword AS (
  SELECT id, row_number() OVER (ORDER BY neg_score, id) AS rank
  FROM (SELECT id, body <@> to_bm25query(:'question', 'docs_body_bm25') AS neg_score
        FROM docs ORDER BY neg_score LIMIT 50) hits WHERE neg_score < 0
), semantic AS (
  SELECT id, row_number() OVER (ORDER BY distance, id) AS rank
  FROM (SELECT id, embedding <=> :'question_embedding' AS distance
        FROM docs WHERE embedding IS NOT NULL
        ORDER BY distance LIMIT 50) nearest
)
SELECT coalesce(k.id, v.id) AS doc_id,
       coalesce(1.0 / (60 + k.rank), 0)
     + coalesce(1.0 / (60 + v.rank), 0) AS rrf
FROM keyword k FULL OUTER JOIN semantic v USING (id)
ORDER BY rrf DESC, doc_id LIMIT 50;
COMMIT;
```

FULL OUTER JOIN keeps either list; COALESCE gives missing ranks zero credit.

Alternative to RRF: blend normalized scores

In each candidate list, map the lowest score to **0** and the highest to **1**. Then take a weighted sum:

```sql
-- Normalize BM25's positive scores; repeat for the vector similarities.
SELECT id, CASE WHEN hi > lo THEN (score - lo) / (hi - lo) ELSE 1 END AS s
FROM (SELECT *, min(score) OVER () AS lo, max(score) OVER () AS hi
      FROM keyword) k;
```

```sql
SELECT coalesce(k.id, v.id) AS doc_id,
       st.w * coalesce(v.s, 0) + (1 - st.w) * coalesce(k.s, 0) AS score
FROM keyword_norm k FULL OUTER JOIN semantic_norm v USING (id)
CROSS JOIN settings st                          -- stored development-set weight
ORDER BY score DESC, doc_id LIMIT 50;           -- candidate pool, before any rerank
```

`w` **weights vectors;** `1 - w` **weights BM25.** Equal scores → 1; missing side → 0.

"Titanic": 50/50 BM25 + bge-small → **Raise the Titanic #1**.

Choose the weight on 31 questions, then freeze it before testing on the other 43.

Rerank: read the clue and each candidate plot

Clue: “wearing white clothes … a trip to a beautiful island … they were killed”

```
question = search.load(conn, "200")
candidates = run_sql_arm(conn, BY_STAGE["rrf_bm25"], question.id)
ranked = search.rerank_lists(conn, question, [candidates])
movies_to_show = ranked.rows[:10]  # trim only after reranking the top 50
```

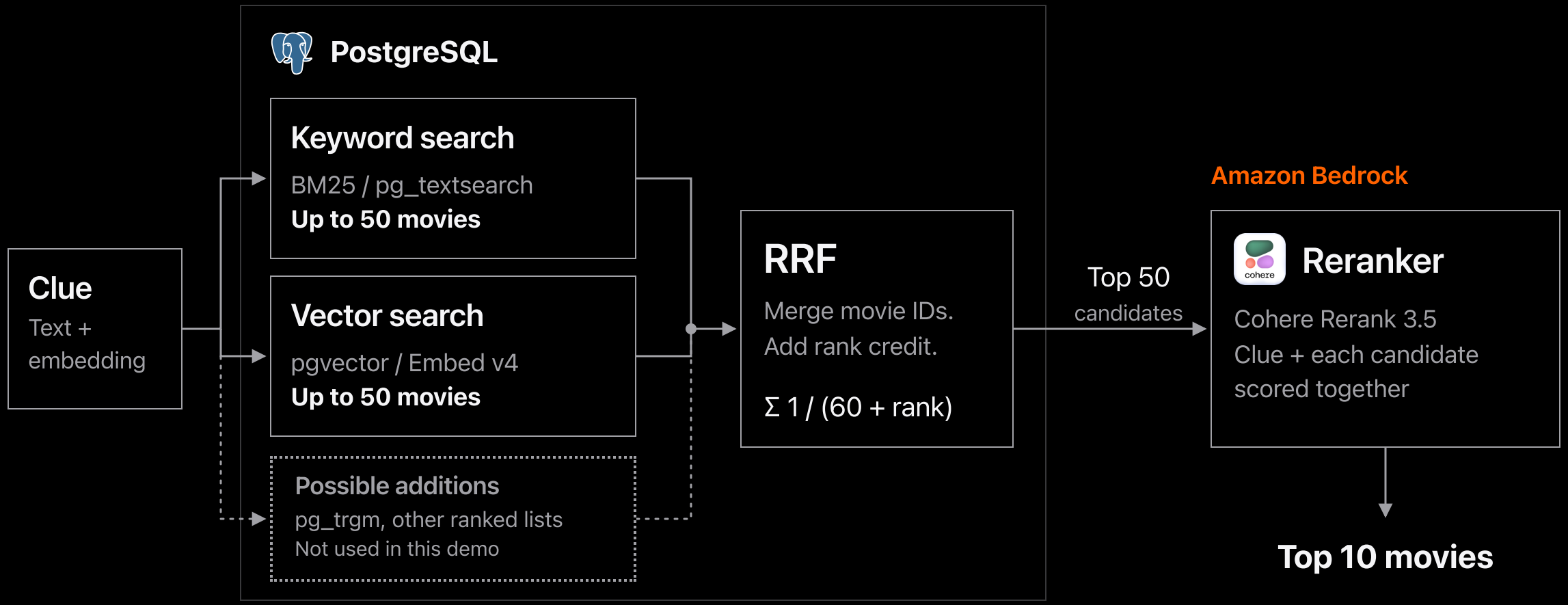
The Island: vector **#18**, RRF **#36**, rerank **#1**.

A **cross-encoder reranker** reads the **query and each candidate together** in one context window to score relevance.
Embedding search encodes them separately, then compares their vectors.

Pitfall: a reranker cannot recover a movie outside its candidate pool.
Median latency: RRF in SQL **62 ms**; the rerank call adds **1.7 s**.

Cohere Rerank 3.5 on Amazon Bedrock: reorder up to 50 candidates, then show 10.

The hybrid search pipeline



Retrieve broadly. Combine ranks. Rerank the shortlist.

Keep up to 50 candidates through reranking; take 10 at the end.

Demo 1: Find the movie

One title.
Then a few clues.

Words → Meaning → Fusion → Rerank

Open the movie demo →

After the demo: what changes when a word is required?





Keep the same movies. Add a required word.

Clue: people in white clothes are promised a trip to an island.

Next: Find similar movies whose search text contains **island**.

Ranking decides what comes first.
Filtering decides what is allowed.

The same CMU movies, using The Island clue and bge-small vectors.

Demo 2: Inspect the filter

Will LIMIT 10 give us ten movies?

Predict the count. Run it. Read the plan.

Open the filter demo →

After the demo: fix the scan and keep the distance order.

```
SELECT id
FROM docs
WHERE tsv @@ plainto_tsquery(
  'english', 'island'
)
ORDER BY embedding_local <=>
  (SELECT embedding_local
   FROM queries
   WHERE id = '200')
LIMIT 10;
```

Clue: “white clothes … a beautiful island … they were killed”

bge-small vectors; required word: **island**

Fix the scan, then restore distance order

HNSW finds 40 candidates → island removes 34 → only 6 of the requested 10 return.

```
BEGIN;
SET LOCAL hnsw.ef_search = 40;              -- same effort as the failed query
SET LOCAL hnsw.iterative_scan = relaxed_order;
WITH nearest AS MATERIALIZED (
  SELECT id, embedding_local <=>
         (SELECT embedding_local FROM queries WHERE id = '200') AS distance
  FROM docs
  WHERE tsv @@ plainto_tsquery('english', 'island')
  ORDER BY distance LIMIT 10
)
SELECT * FROM nearest ORDER BY distance + 0;
COMMIT;
```

Now: 10 rows. Continue scanning, then sort the returned candidates.

`MATERIALIZED` + `distance + 0` ensures a fresh sort on PostgreSQL 18.

Inspect with EXPLAIN (ANALYZE, BUFFERS, SETTINGS, TIMING OFF). Scan budgets can still limit results.

Reuse RRF through a PostgreSQL function

Main: `hybrid_search()` combines BM25 + Embed v4 ranks with RRF.
Alternative: `hybrid_search_blend()` combines normalized BM25 + bge-small scores.
Call below: The Island clue + its Embed v4 vector → five movies matching "island".

```
SELECT doc_id, score, keyword_rank, vector_rank
FROM hybrid_search(:'question', :'question_embedding',
                   match_count => 5, required_terms => 'island')
ORDER BY score DESC, doc_id;
```

SETTINGS STORED IN THE FUNCTION	APPLIED ON EVERY CALL
`hnsw.ef_search = 100`	Search effort for the vector top-50 list
`hnsw.iterative_scan = relaxed_order`	Keep scanning when the filter rejects movies
`plan_cache_mode = force_custom_plan`	Plan using this call's actual filter arguments

One row per movie: ID, RRF score, BM25 rank and vector rank.
Required terms filter **both candidate lists, before their LIMIT**.

Keep metadata boosts bounded

```
WITH pool AS (
  SELECT h.doc_id, h.score, coalesce(m.box_office, 0) AS box_office
  FROM hybrid_search_blend(:'question', :'question_embedding_local', match_count => 50) h
  LEFT JOIN film_meta m ON m.doc_id = h.doc_id
), boosted AS (
  SELECT *, score * (1 + 0.25 * box_office::float8
                     / nullif(max(box_office) OVER (), 0)) AS boosted
  FROM pool
)
SELECT doc_id, coalesce(boosted, score) AS score FROM boosted
ORDER BY score DESC, doc_id LIMIT 10;
```

RANK	"TITANIC"	BOX OFFICE		SCORE	
1	Titanic 1997	$2.19B	× 1.25	1.118	▲ 1
2	Raise the Titanic 1980	$7.0M	× 1.001	0.935	▼ 1
3	Titanic 1953	$2.3M	× 1.000	0.893	

Score blend (BM25 + bge-small) → boost by up to 25% → show 10 of 50 candidates.

Pitfall: popularity is a preference, not a relevance judgment. A title alone still leaves intent open.

Measure the ladder on all 43 test questions

CAPABILITY	NDCG@10	RECALL@50
BM25	7.9	30.2
Vector (Embed v4) baseline	29.8	51.2
BM25 + Embed v4 (RRF)	25.9	48.8
BM25 + Embed v4 (tuned blend)	32.6	51.2
RRF + rerank	40.3	48.8

RRF won the Wolfhound example, but on average it scores below vector alone.

NDCG@10: rewards the known movie near the top. **Recall@50:** percentage of questions whose answer is in the top 50. Both are shown on a 0–100 scale; higher is better.
Reranking reorders the same candidates, so Recall@50 stays at 48.8.

What this collection can and cannot prove

- **One English movie collection.** 7,531 movies and 43 test questions; selected examples explain mechanisms.
- **One known movie per question.** Useful alternatives may be unjudged; the task is title recovery.
- **Measure end-to-end latency.** Fresh questions need embedding and reranking calls; SQL timings alone are incomplete.
- **Performance depends on the plan.** Small corpora, model dimensions, cache state and filters change the access path.

Keep the ladder. Re-measure each step on your own questions.

The agent chooses how to search

A **tool** is a Python function the agent can call with named inputs. Here, it runs an existing PostgreSQL search.

```
Agent
  → search_movies(query, method="hybrid")  [Python tool]
  → hybrid_search(query, embedding)       [PostgreSQL function: RRF]
  → ranked movies + each arm's contribution
```

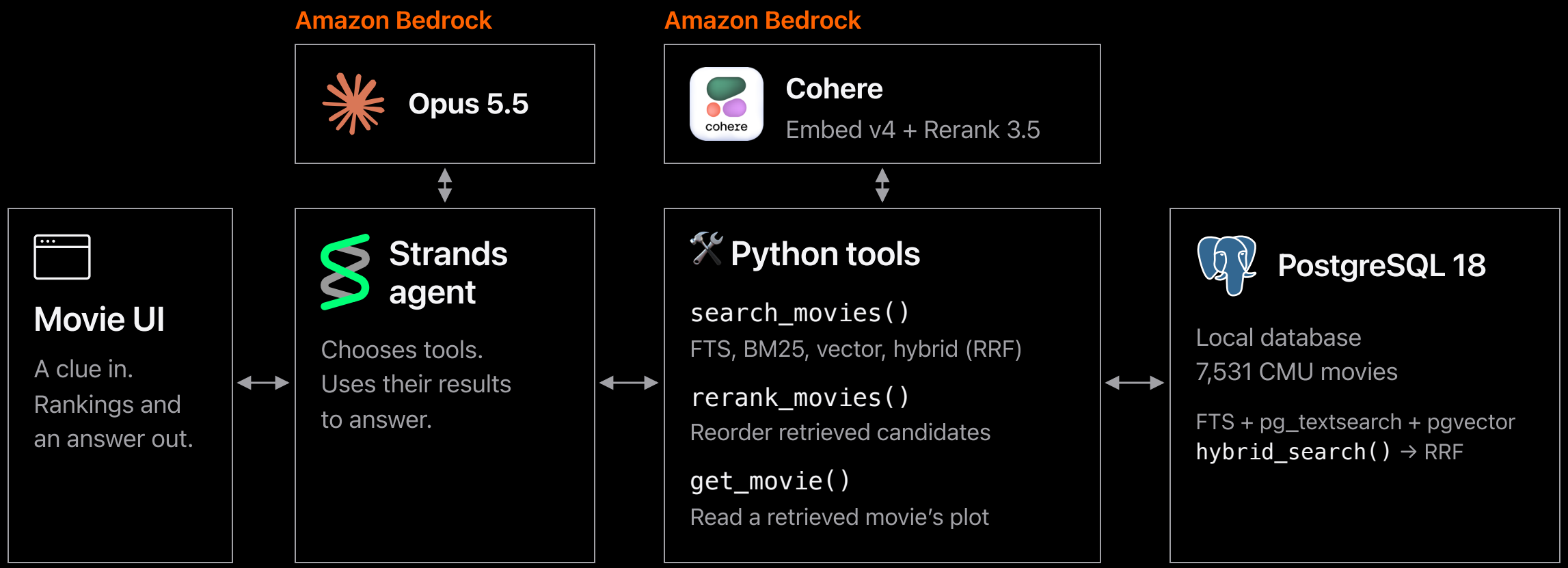
`hybrid` means BM25 + Embed v4 combined by RRF.

- `search_movies()` also accepts `fts`, `bm25` or `vector` to run one method alone.
- `rerank_movies()` reorders the candidates; `get_movie()` reads a returned plot.

Pitfall: a fluent answer still depends on what retrieval found.

The same 7,531 CMU movies. Next: connect the tools to an agent.

How the agent uses PostgreSQL



The agent chooses the tools. PostgreSQL ranks the movies.

Prepared clues reuse stored embeddings and reranks; the agent's conversation is live.

Demo 3: Let the agent search

“A hero with a pet bat…”

Watch the tool calls and rank changes.
Ask why Wolfhound moved up.

Open pg_sherlock →

After the demo: three things to take home.



Meet pg_sherlock 🔍✨

`search_movies()` finds candidates.
`rerank_movies()` reorders them.
`get_movie()` reads the plot.

Three things to take home

- **Start with strong baselines.**
  BM25 for words, vectors for meaning. Required matches belong in filters.
- **Measure whether fusion helps.**
  Choose settings on one set of questions; measure on another. Check uncertainty before claiming a gain.
- **Read the PostgreSQL plan.**
  Inspect actual rows, filters and index paths. Keep the simplest query that meets quality and latency needs.

Take the SQL and the evaluation home

Build: SQL for BM25, vectors, RRF, filters, and bounded boosts.

Inspect: the demo, the executed query, and its plan.

Decide: compare against your baselines on independently judged questions.

Materials will be published after the conference.



github.com/shayons/talks

Thank You!



May your first result be the one they meant.



Shayon Sanyal

Connect on LinkedIn

linkedin.com/in/shayonsanyal

References

- **PostgreSQL 18:** Full-text search, Using EXPLAIN, auto_explain
- **pgvector 0.8.6:** HNSW, filtering, iterative index scans; **pg_textsearch 1.4.0:** BM25 for PostgreSQL
- **RRF:** Cormack, Clarke & Büttcher, SIGIR 2009; **HNSW:** Malkov & Yashunin, arXiv:1603.09320
- **Score blending:** Bruch, Gai & Ingber, "An Analysis of Fusion Functions for Hybrid Retrieval", ACM TOIS 2023; **bge-small-en-v1.5:** BAAI, MIT license, run with fastembed
- **Movies:** CMU Movie Summary Corpus + TREC Tip-of-the-Tongue
- **SQL, demo and evaluation:** Talk repository (available after the conference)

How the filter changes the plan

Requested: 10. With iterative scanning off: 40 candidates – 34 rejected by “island” = **6 returned**.

```
iterative_scan = off
  Index Scan using docs_embedding_local_hnsw on docs
    Filter: (tsv @@ 'island'::tsquery)
    actual rows=6    Rows Removed by Filter: 34

iterative_scan = relaxed_order
  Index Scan using docs_embedding_local_hnsw on docs
    Filter: (tsv @@ 'island'::tsquery)
    actual rows=10   Rows Removed by Filter: 72
```

409 of 7,531 movies match island. bge-small with `ef_search = 40`.

Re-sort relaxed results by distance.

For **princess**, this planner instead chooses **GIN + an exact sort**.

SQL latency: read the plan

QUERY, 43 TEST QUESTIONS	PLAN USED	P50	P95
FTS OR + `ts_rank_cd`	GIN, then ranks every match	695 ms	1,778 ms
BM25 (pg_textsearch)	BM25 index	26 ms	34 ms
Vector, bge-small (384)	HNSW	1.4 ms	1.7 ms
Vector, Embed v4 (1,536)	**Seq Scan**, planner's choice	37 ms	39 ms
Same query, `enable_seqscan = off`	HNSW, 95% same top 50	3.7 ms	4.2 ms
RRF: BM25 + Embed v4	Seq Scan + BM25 index	62 ms	73 ms
Cohere Rerank 3.5 call	Amazon Bedrock	1.7 s	2.4 s

Vector: the exact scan reads **50,308** buffers; HNSW reads **2,070**. The large vectors are stored outside the main row (TOAST).
FTS: a long clue's OR query matches **7,119 of 7,531** movies on average. Ranking, not GIN, takes the time.

p50: median time. **p95:** 95% of measured times are at or below this value. Lower is faster.
SQL: EXPLAIN ANALYZE time; rerank: API-call time. Warm cache, one laptop; plans depend on the workload.