Hybrid Search

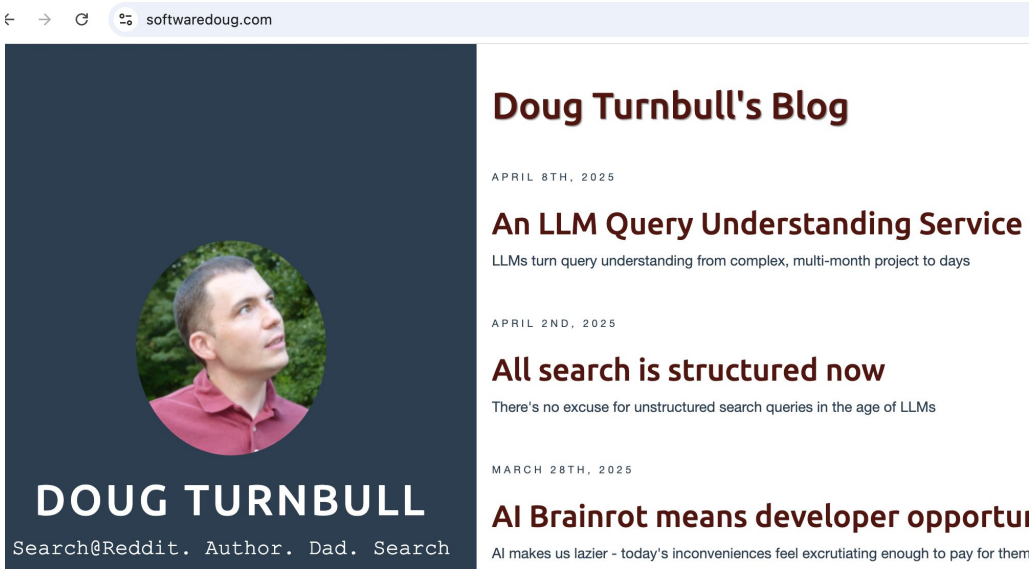
Optimizing the R in RAG

Apr 16, 2025

Obligatory Bio Slide

👋 Hi I'm Doug
(@softwaredoug everywhere)

I blog here: http://softwaredoug.com



Obligatory Plug

AI Evals For Engineers

NEW · 4 WEEKS · COHORT-BASED COURSE

Learn proven approaches for quickly improving AI applications. Build AI that works better than the competition, regardless of the use-case.

This course is popular
22 people enrolled last week.

HOSTED BY

Hamel Husain and Shreya Shankar
ML Engineers who've spent 25+ combined years building & evaluating AI systems.

Cheat at Search with LLMs

NEW · 4 WEEKS · COHORT-BASED COURSE

Vibe-code your way to AI-Powered Search

This course is popular
9 people enrolled last week.

HOSTED BY

Doug Turnbull
Led Search Reddit + Shopify. Wrote Relevant Search + AI Powered Search

Obligatory Plug
https://maven.com/softwaredoug/cheat-at-search
Discount Code: **searchybird** good through Apr

Can’t cover in 45 mins…

1. How lexical search actually works (ask chat GPT about: inverted index, read “Relevant Search” 😉)

2. What is an embedding

3. Lexical scoring, vector scoring (cosine, euclidean, etc similarities) etc

   Intuitive sense of “close” good enough for today :)

Also won’t cover

1. RRF - Reciprocal Rank Fusion

RRF is Not Enough

NOVEMBER 3RD, 2024

Hybrid search means combining lexical and vector search results into one result listing.

“We’ll just use Reciprocal Rank Fusion” I’m sure I’ve said from time to time.

As if RRF is kind of “a miracle occurs”. You get the best of both worlds, and suddenly your search looks incredible.

Take the query `hello to the planet`. Let’s say we start with reasonable results from a vector search system (follow along in this notebook)

vector_sim	texts	vector_rank

Assumption: embeddings good first pass search

Embeddings get you *close* but not all the way

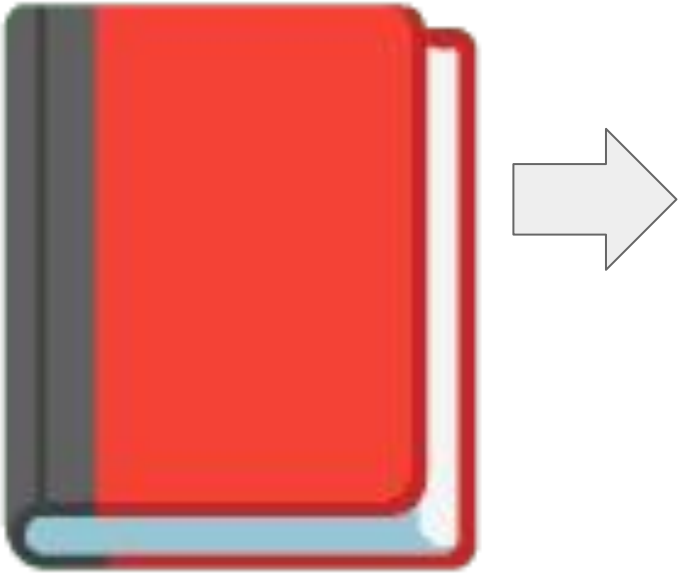
ID	Title	Vector (256? 512? Or more dimensions)
0	mary had a little lamb	[0.9, 0.8, -0.5, 0.75, ..]
1	mary had a little ham	[0.6, 0.4, -0.4, 0.60, ..]
2	a little ham	[-0.2, 0.5, 0.9, -0.45, ..]
3	little mary had a scam	[0.4, -0.5, 0.25, 0.14, ..]
4	ham it up with mary	[0.2, 0.5, 0.2, 0.45, ..]
5	Little red riding hood had a baby sheep?	[0.95, 0.79, -0.49, 0.65, ..]

Similar!

(despite sharing few terms)

Chunked

You've chunked your data into a meaningful "search document" with important metadata:



```
{
    "Book_title": "Nursery Rhymes"
    "Section": "Mary Had a Little
    Lamb"
    "Text": "..."

}
```

Embedding for *whole document*

We want an embedding capturing as much of the document as is reasonable

```
text_concatted = data['product_name'] + ' -- ' + data['product_description']

embedding = model.encode(text_concatted)
```

(Not just a title embedding)

Embedding is ~ two-towerable

Short text (ie queries) and long text (paragraphs) can be mapped in similarity space

QUERY: Kid story about sheep

Document:

Mary had a little lamb, little lamb, little lamb.

Mary had a little lamb, its fleece was white as snow.

And everywhere that Mary went. Mary went. Mary went.

And everywhere that Mary went, the lamb was sure to go.

It followed her to school one day, school one day, school one

Similar

Bonus: embedding is a two tower model!

Query Features

- Query embedding
- Query

Document Features

- Name
- Description
- Product image embedding
- ???

(Biencoder, learned on labeled data)

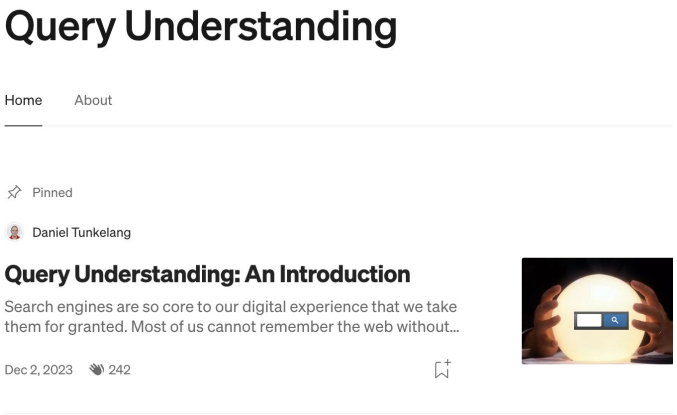
After embedding we boost/rerank/…

Exact name match?

- Move these to the top!

Query mentions color?

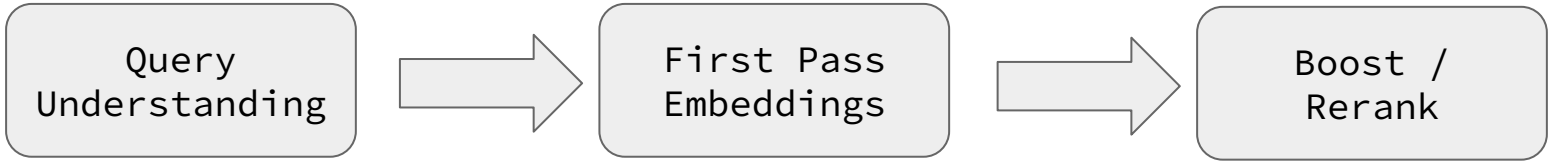
- Ensure color matches boosted



(Different query types == different treatments!)

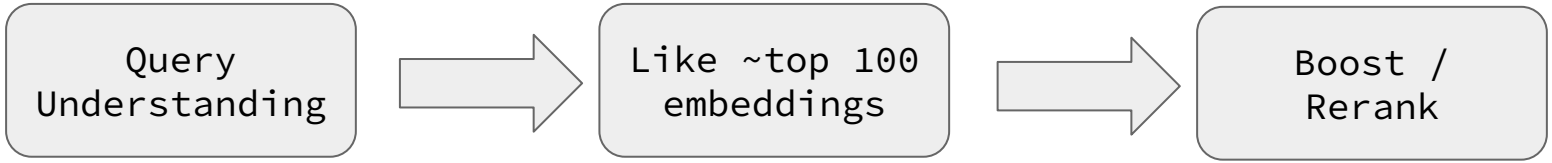
http://queryunderstanding.com

Ideal:

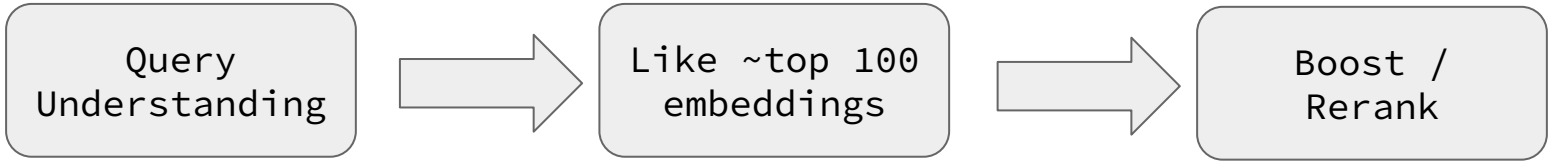


(depending on needs of query)

Reality:

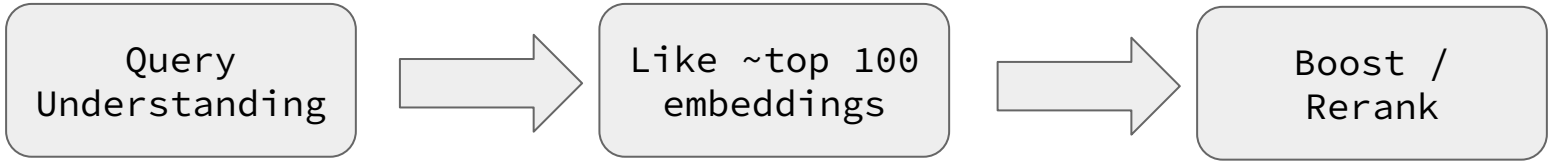


Reality:



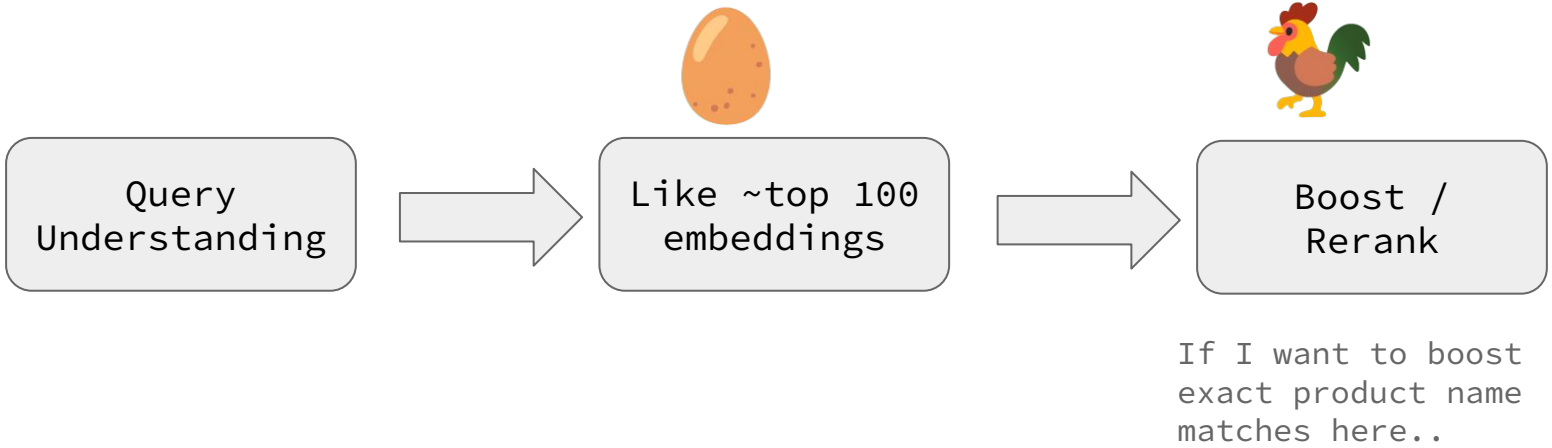
(Do we have the right top 100 to boost?)

Reality:



Need to filter this to
“good” 100 or so

Chicken and egg problem:



Chicken and egg problem:

~2021 vector DB

No WHERE!

SELECT * FROM <search_engine>

👎 Can’t guarantee product name matches promoted

ORDER BY vector_similarity(query_embedding, title_embedding)
LIMIT 100

2025 vector DB (search engine)

```
SELECT * FROM <search>

WHERE [trowel] in product_name


     ...

ORDER BY vector_similarity(query_embedding, title_embedding)
LIMIT 100
```

BEFORE vector_similarity
Get candidates matching
“trowel”

👍 Now I have matches!

~2025 era vector DB (search engine)

```
SELECT * FROM <search>

WHERE [trowel] in product_name

    ...

ORDER BY vector_similarity(query_embedding, title_embedding)
LIMIT 100
```

BEFORE vector_similarity
Get candidates matching
“mary”

🚨 *How does your vector DB **pre**-filter? Can you do this at scale?*

… and “where” could be *anything*

Search for “garden trowel”

```
SELECT * FROM <search>

WHERE “lawn_and_garden” in department

     AND “trowel” in item_type

     AND (garden in title OR garden in description OR

          trowel in title OR trowel in description)

ORDER BY vector_similarity(query_embedding, title_embedding)
LIMIT 100
```

Somehow we turn the query to this dept / item type

… and “where” could be *anything*

Search for “garden trowel”

```
SELECT * FROM <search>

WHERE “lawn_and_garden” in department

     AND “trowel” in item_type

     AND (garden in title OR garden in description OR

          trowel in title OR trowel in description)

ORDER BY vector_similarity(query_embedding, title_embedding)
LIMIT 100
```

And also match query terms in tokenized title/description

… and “where” could be *anything*

Search for “garden trowel”

```
SELECT * FROM <search>

WHERE “lawn_and_garden” in department

     AND “trowel” in item_type

     AND (garden in title OR garden in description OR

          trowel in title OR trowel in description)

ORDER BY vector_similarity(query_embedding, title_embedding)
LIMIT 100
```

And also match query terms

(yes you search nerds, I’m ignoring BM25 and lexical scoring for now)

Practically: there’s a vector index

We can reasonably get top K...

Search for “garden trowel”

```
SELECT * FROM <search>

WHERE “lawn_and_garden” in department

     AND “trowel” in item_type

     AND (garden in title OR garden in description OR

          trowel in title OR trowel in description)

ORDER BY vector_similarity(query_embedding, title_embedding)
LIMIT 100
```

Get top 100 from this set via an index

(otherwise we scan all results to score them)

There’s more than one “top K” we care about

What about “pure” vector matches?

```
SELECT * FROM <search>

WHERE “lawn_and_garden” in department

     AND “trowel” in item_type

     AND (garden in title OR garden in description OR

          trowel in title OR trowel in description)

ORDER BY similarity(query_embedding, title_embedding)
LIMIT 100
```

100 from this set

```
UNION ALL

SELECT * FROM <search>

WHERE “lawn_and_garden” in department

     AND “trowel” in item_type

ORDER BY similarity(query_embedding, title_embedding)
LIMIT 100
```

There’s more than one candidate set

What about “pure” vector matches?

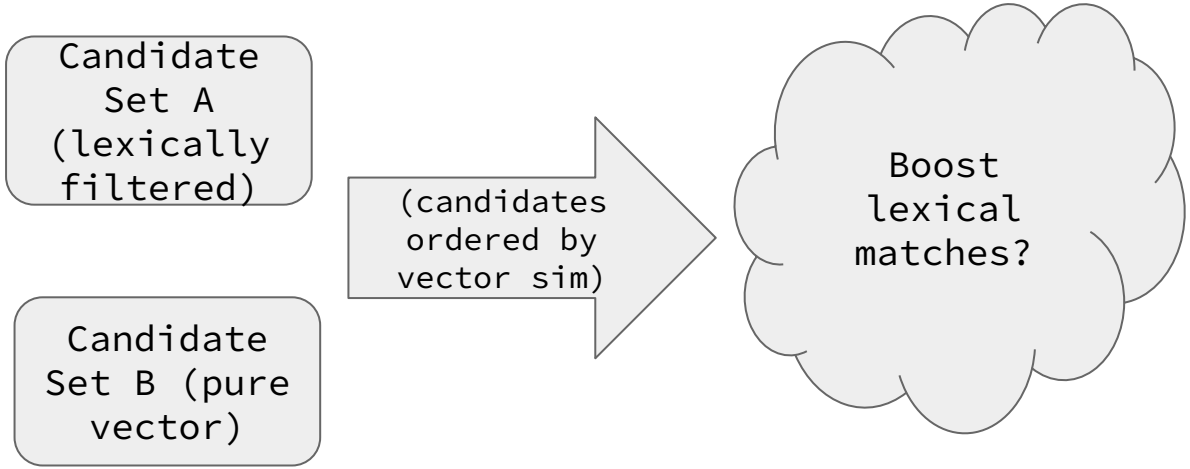
```
SELECT * FROM <search>

WHERE “lawn_and_garden” in department

     AND “trowel” in item_type

     AND (garden in title OR garden in description OR

          trowel in title OR trowel in description)

ORDER BY similarity(query_embedding, title_embedding)
LIMIT 100
```

UNION ALL

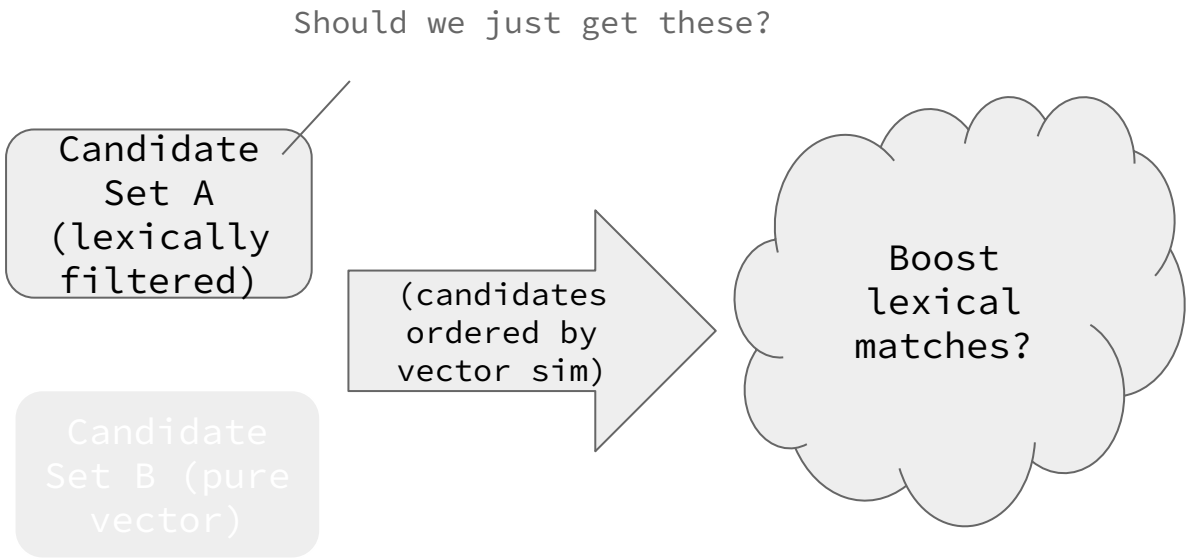
+ 100 from this set

```
SELECT * FROM <search>

WHERE “lawn_and_garden” in department

     AND “trowel” in item_type

ORDER BY similarity(query_embedding, title_embedding)
LIMIT 100
```

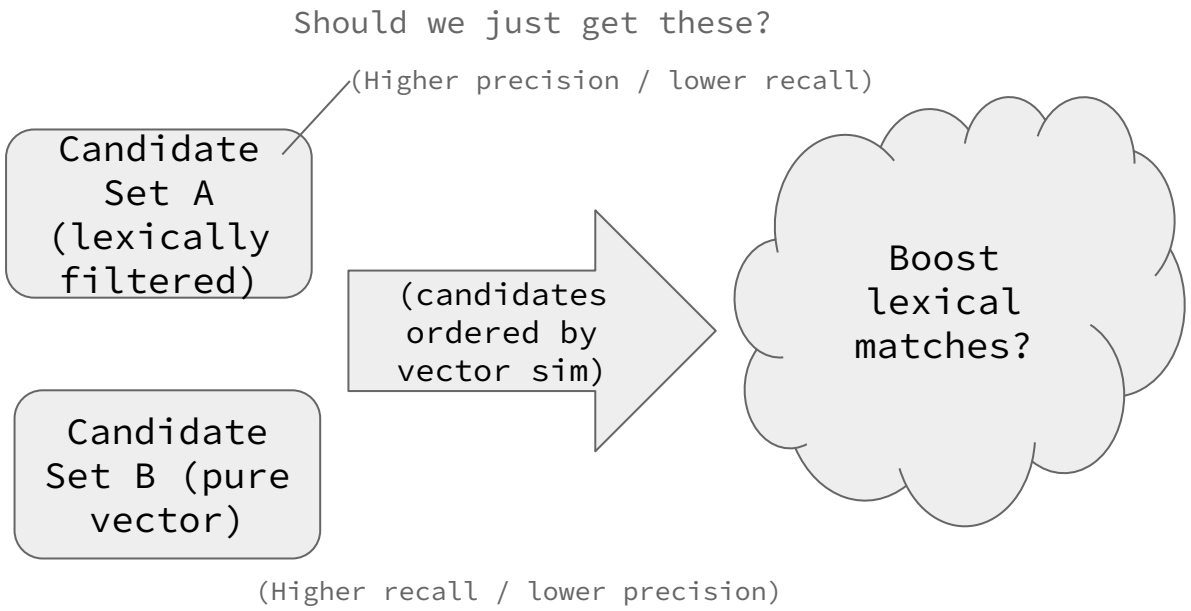
With squiggly lines…



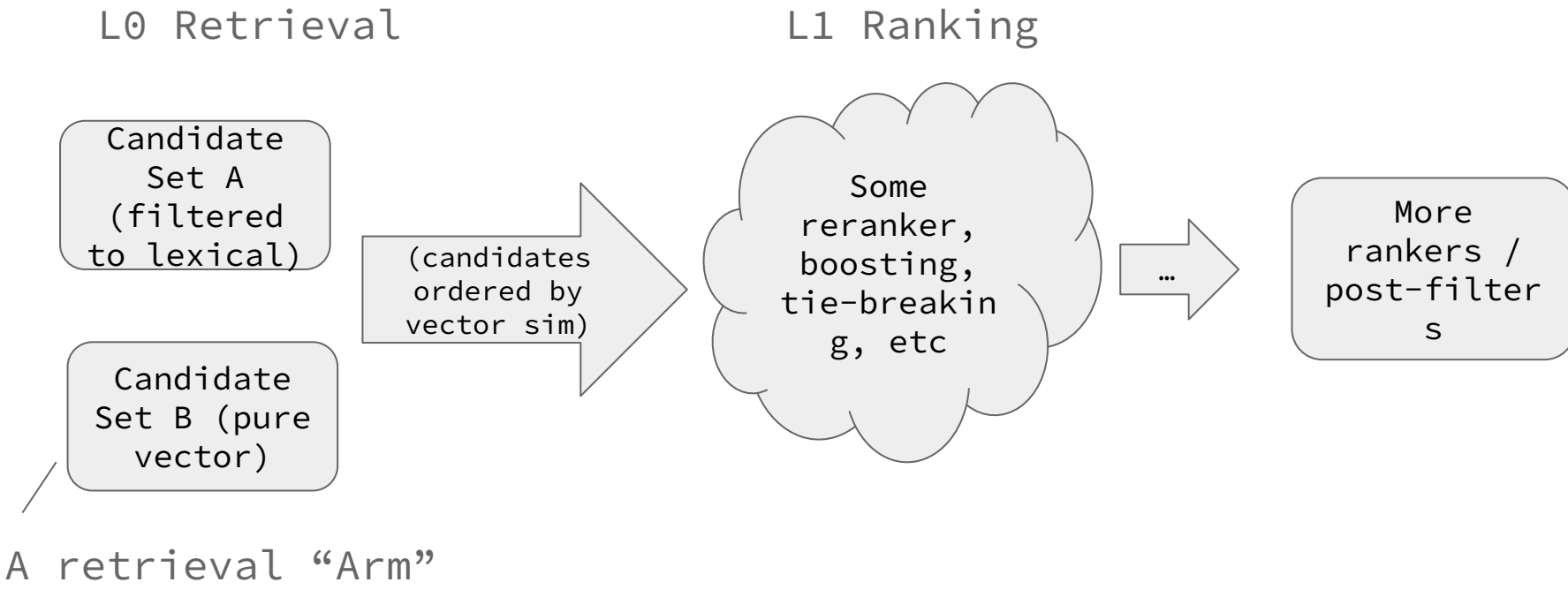
Why do we do it this way?



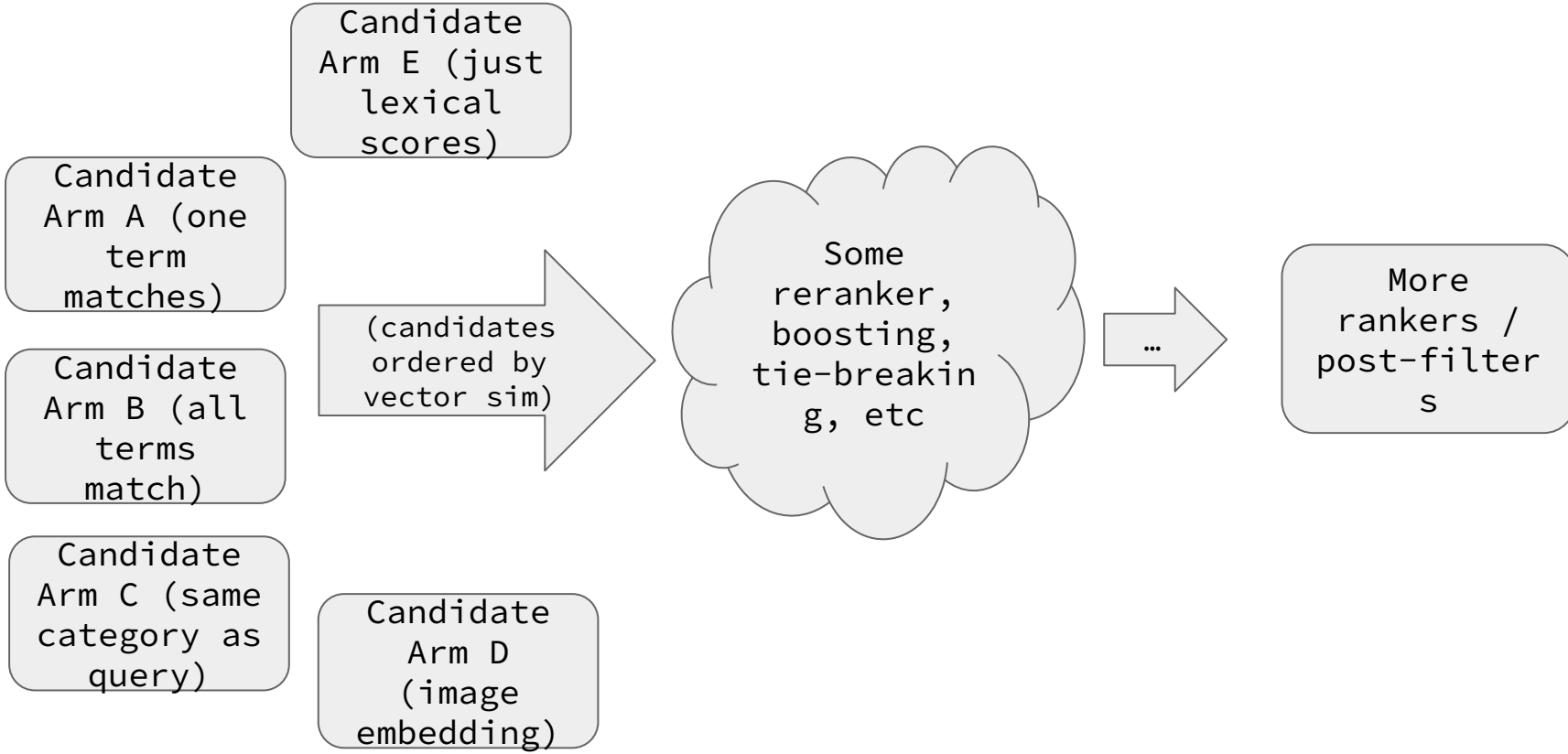
Why do we do it this way?



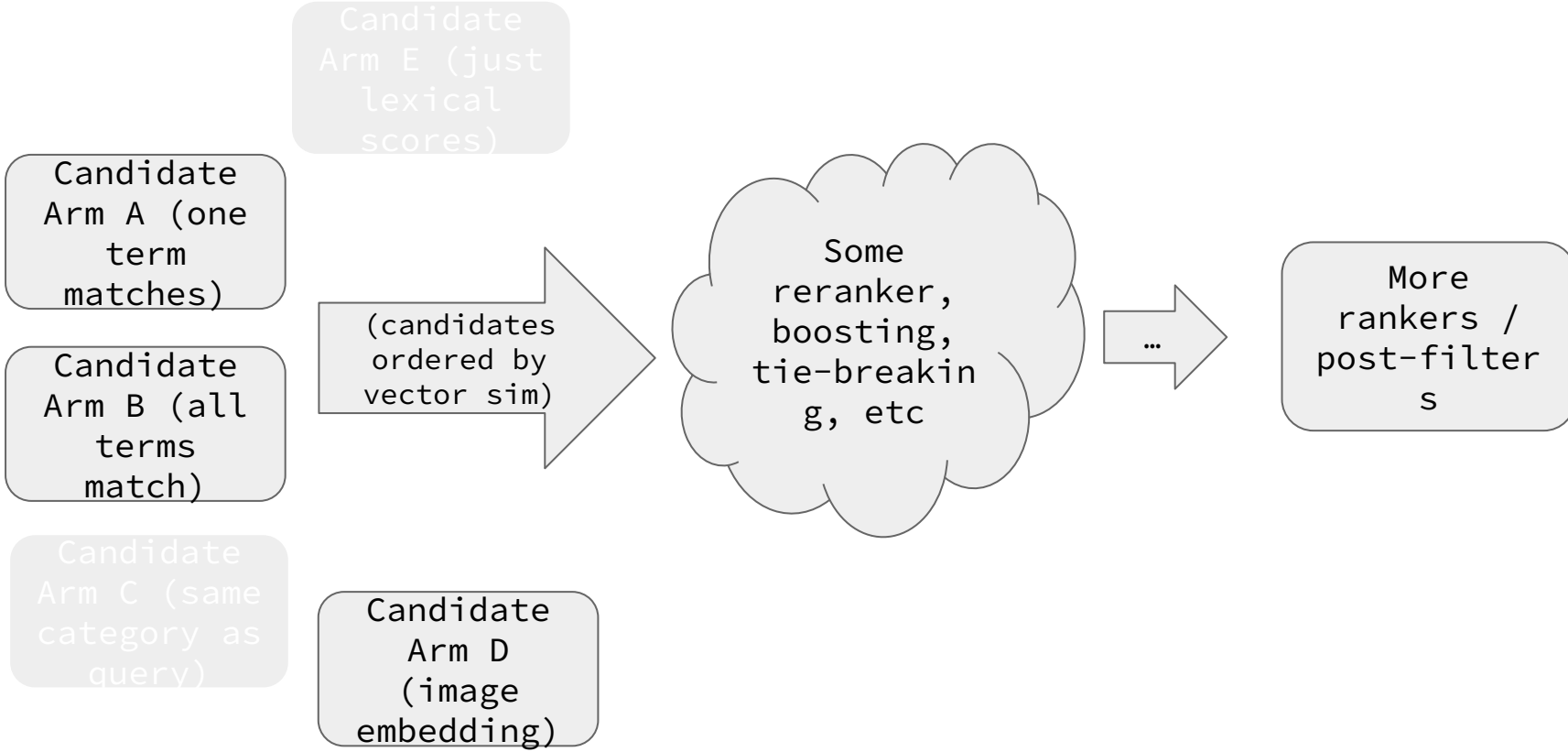
With squiggly lines…



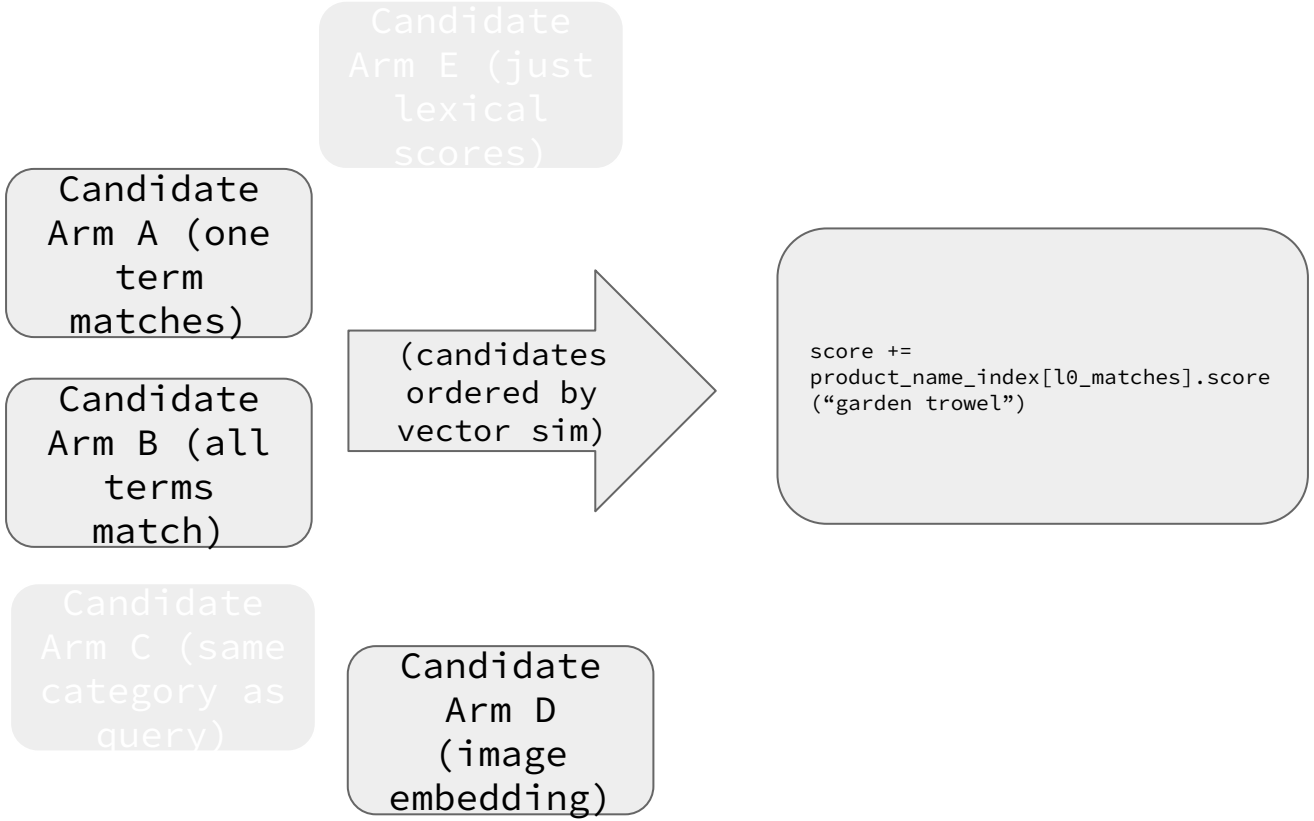
And many retrieval arms



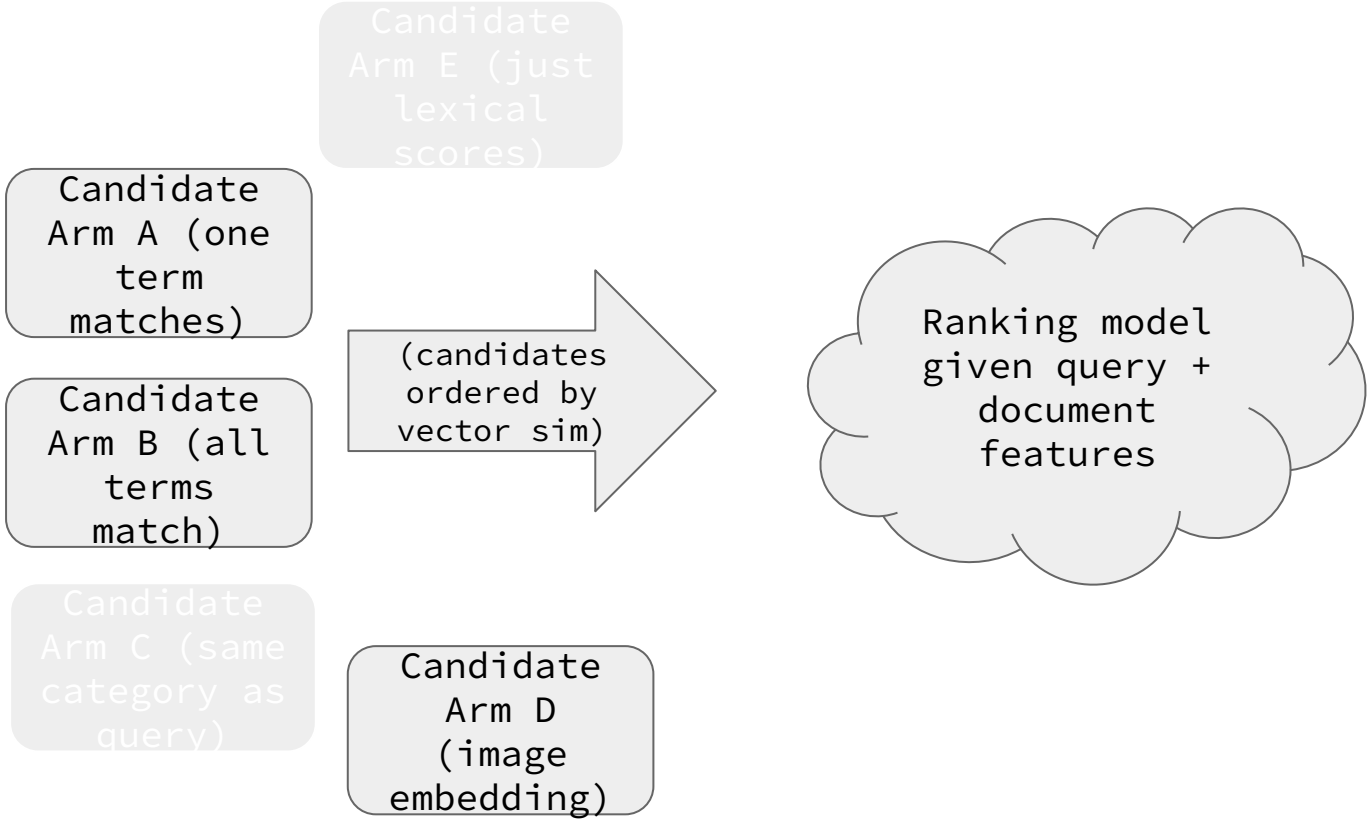
Or depending on the query



Then the boost



Or a model


Candidate Arm E (just lexical scores)
Candidate Arm A (one term matches)
Candidate Arm B (all terms match)
Candidate Arm C (same category as query)
Candidate Arm D (image embedding)
(candidates ordered by vector sim)
Ranking model given query + document features

That’s the theory at least

https://colab.research.google.com/drive/1HmWdKON-wxHMQCnig0hVA3u0-OX1I2Ph