

# Setup



# How to Set Up the HyBench Local Database

## Phase 1: Software Installation

1. **SQL Server 2025 (Preview):** Download and install the **Developer Edition**.
  - o *During install:* Select "Basic" installation. Note the server name (usually localhost or YourPCName\MSSQLSERVER).
2. **SSMS (SQL Server Management Studio):** Download the latest preview or stable version (v20 or v21).
3. **Python:** Install Python 3.x.
4. **Python Libraries:** Open a terminal (cmd/powershell) and run:
5. Bash

```
pip install pyodbc pandas
```

6.

7.

## Phase 2: Folder & Permissions (Crucial Step!)

*If you skip this, the database will fail to load files.*

1. Go to your C:\ drive.
2. Create a folder named **TempData**.
3. **Get the 6 CSV files** from Vedant (via USB/Drive) and paste them inside C:\TempData.
  - o *Make sure they are named:* page\_200000.csv, page\_extra\_200000.csv, page\_embedding\_200000.csv, text\_200000.csv, text\_embedding\_200000.csv, revision\_clean\_200000.csv.
4. **Grant Permissions:**
  - o Right-click C:\TempData -> **Properties** -> **Security**.
  - o Click **Edit** -> **Add**.
  - o Type **NT SERVICE\MSSQLSERVER** -> Click **Check Names** -> **OK**.
  - o Check "Allow" for **Read & execute**.
  - o Click **OK**.

## Phase 3: Initialize Database (Run in SSMS)

Open SSMS, connect to your server, and run this script to create the database and tables.

SQL

```
USE master;
```

```
GO
```

```
CREATE DATABASE index_hybench_100k;

GO

USE index_hybench_100k;

GO

-- 1. Create Final Tables

CREATE TABLE dbo.page (
    page_id INT NOT NULL PRIMARY KEY CLUSTERED,
    page_title NVARCHAR(MAX) NOT NULL,
    page_is_redirect BIT, page_is_new BIT,
    page_touched NVARCHAR(50), page_links_updated NVARCHAR(50),
    page_latest BIGINT, page_len BIGINT,
    page_embedding VECTOR(384)
);

CREATE TABLE dbo.text (
    old_id INT NOT NULL PRIMARY KEY CLUSTERED,
    old_text NVARCHAR(MAX) NOT NULL,
    old_flags NVARCHAR(100),
    text_embedding VECTOR(384)
);

CREATE TABLE dbo.revision (
    rev_id INT NOT NULL PRIMARY KEY CLUSTERED,
    rev_page INT NOT NULL, rev_text_id INT NOT NULL,
    rev_timestamp NVARCHAR(50), rev_minor_edit BIT,
    rev_user_text NVARCHAR(255), rev_comment NVARCHAR(MAX)
```

```
);
```

```
-- 2. Create Staging Tables
```

```
CREATE TABLE dbo.stage_page (page_id NVARCHAR(MAX), page_title  
NVARCHAR(MAX));
```

```
CREATE TABLE dbo.stage_page_extra (page_len NVARCHAR(MAX), page_touched  
NVARCHAR(MAX), page_namespace NVARCHAR(MAX));
```

```
CREATE TABLE dbo.stage_page_embedding (embedding_json NVARCHAR(MAX));
```

```
CREATE TABLE dbo.stage_text (old_id NVARCHAR(MAX), old_text NVARCHAR(MAX));
```

```
CREATE TABLE dbo.stage_text_embedding (embedding_json NVARCHAR(MAX));
```

```
CREATE TABLE dbo.stage_revision (rev_id NVARCHAR(MAX), rev_page_id  
NVARCHAR(MAX), rev_minor_edit NVARCHAR(MAX), rev_actor NVARCHAR(MAX),  
rev_timestamp NVARCHAR(MAX));
```

```
GO
```

```
-- 3. Bulk Load CSVs to Staging
```

```
PRINT 'Loading CSVs...';
```

```
BULK INSERT dbo.stage_page FROM 'C:\TempData\page_200000.csv' WITH (FORMAT =  
'CSV', FIELDTERMINATOR = ',', ROWTERMINATOR = '0x0a', TABLOCK);
```

```
BULK INSERT dbo.stage_page_extra FROM 'C:\TempData\page_extra_200000.csv' WITH  
(FORMAT = 'CSV', FIELDTERMINATOR = ',', ROWTERMINATOR = '0x0a', TABLOCK);
```

```
BULK INSERT dbo.stage_page_embedding FROM  
'C:\TempData\page_embedding_200000.csv' WITH (FORMAT = 'CSV', FIELDTERMINATOR  
= ',', ROWTERMINATOR = '0x0a', TABLOCK);
```

```
BULK INSERT dbo.stage_text FROM 'C:\TempData\text_200000.csv' WITH (FORMAT =  
'CSV', FIELDTERMINATOR = ',', ROWTERMINATOR = '0x0a', TABLOCK);
```

```
BULK INSERT dbo.stage_text_embedding FROM  
'C:\TempData\text_embedding_200000.csv' WITH (FORMAT = 'CSV', FIELDTERMINATOR  
= ',', ROWTERMINATOR = '0x0a', TABLOCK);
```

```
BULK INSERT dbo.stage_revision FROM 'C:\TempData\revision_clean_200000.csv' WITH  
(FORMAT = 'CSV', FIELDTERMINATOR = ',', ROWTERMINATOR = '0x0a', TABLOCK);
```

```
GO  
PRINT 'Staging loaded.';
```

#### Phase 4: Load Final Data (Run Python)

Create a file named setup\_data.py and run it. This handles the complex JSON-to-Vector conversion for the first 100k rows.

Python

```
import pyodbc  
  
import pandas as pd  
  
import time  
  
  
# --- EDIT THIS IF NEEDED ---  
  
YOUR_SERVER_NAME = "localhost"  
  
# -----  
  
  
conn_str = f"DRIVER={{ODBC Driver 17 for SQL  
Server}};SERVER={YOUR_SERVER_NAME};DATABASE=index_hybench_100k;Trusted_C  
onnection=yes;"  
  
ROWS = 100000  
  
  
  
def run_load():  
  
    conn = pyodbc.connect(conn_str)  
  
    cursor = conn.cursor()  
  
    cursor.fast_executemany = True  
  
  
    print("1. Loading Page Table...")
```

```

df = pd.read_sql(f"SELECT TOP {ROWS} * FROM dbo.stage_page p JOIN
dbo.stage_page_extra pe ON 1=1 JOIN dbo.stage_page_embedding pem ON 1=1 WHERE
p.page_title IS NOT NULL", conn)

# Note: The JOIN above is simplified for the python script logic we used previously.

# Actually, use the join logic below for accuracy:

df_p = pd.read_sql(f"SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL))
rn FROM dbo.stage_page", conn).head(ROWS)

df_e = pd.read_sql(f"SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL))
rn FROM dbo.stage_page_extra", conn).head(ROWS)

df_em = pd.read_sql(f"SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL))
rn FROM dbo.stage_page_embedding", conn).head(ROWS)

df = df_p.merge(df_e, on="rn").merge(df_em, on="rn")



data = []

for _, r in df.iterrows():

    data.append((int(r['page_id']), r['page_title'] or "", 0, 0, r['page_touched'] or None,
r['page_touched'] or None, None, int(r['page_len'] or 0), r['embedding_json']))

cursor.executemany("INSERT INTO dbo.page VALUES (?,?,?,?,?,?,?,?,?)", data)

conn.commit()

print("2. Loading Text Table...")

df_t = pd.read_sql(f"SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL))
rn FROM dbo.stage_text", conn).head(ROWS)

df_tm = pd.read_sql(f"SELECT *, ROW_NUMBER() OVER (ORDER BY (SELECT NULL))
rn FROM dbo.stage_text_embedding", conn).head(ROWS)

df = df_t.merge(df_tm, on="rn")


data = []

for _, r in df.iterrows():

```

```

        data.append((int(r['old_id']), r['old_text'] or "utf-8", r['embedding_json']))

    cursor.executemany("INSERT INTO dbo.text VALUES (?,?,?,?)", data)

    conn.commit()

print("3. Loading Revision Table...")

df = pd.read_sql(f"SELECT TOP {ROWS} * FROM dbo.stage_revision", conn)

data = []

for _, r in df.iterrows():

    data.append((int(r['rev_id']), int(r['rev_page_id']), int(r['rev_id']), r['rev_timestamp'],
                int(r['rev_minor_edit'] or 0), r['rev_actor'], None))

    cursor.executemany("INSERT INTO dbo.revision VALUES (?,?,?,?,?,?)", data)

    conn.commit()

print("Done!")

```

```

if __name__ == "__main__":
    run_load()

```

## Phase 5: Create Indexes (Run in SSMS)

Run this final script to build the vector indexes.

SQL

```

USE index_hybench_100k;

GO

ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;

GO

```

```
CREATE VECTOR INDEX [idx_page_embedding] ON dbo.page(page_embedding) WITH  
(METRIC = 'COSINE', TYPE = 'DISKANN');  
  
CREATE VECTOR INDEX [idx_text_embedding] ON dbo.text(text_embedding) WITH  
(METRIC = 'COSINE', TYPE = 'DISKANN');  
  
GO  
  
PRINT 'Complete!';
```

---

## 🔍 Part 2: Query 17 (IQ2) Translation

Query:

```
SELECT old_id, old_text FROM text WHERE text_embedding {op} '{q}' BETWEEN {d} AND  
{d*} ...
```

### Explanation

This is a Bounded Range Search (or Annulus Search).

It asks for all text vectors that fall within a specific distance band (e.g., "distance is greater than 0.2 but less than 0.5").

### Indexed Translation: Not Possible

Just like Query 2 and Query 6, this is a range search.

- VECTOR\_SEARCH (the index tool) requires a TOP\_N limit ("Get me 10 items").
- It **cannot** handle "Get me items where distance is between X and Y".
- Therefore, we **cannot** use the index. We must use VECTOR\_DISTANCE in the WHERE clause, which forces a full table scan.

### IQ2 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Use your 100k DB
```

GO

```
-- 1. Define parameters
```

```
DECLARE @dist_min FLOAT = 0.2; -- {d}
```

```
DECLARE @dist_max FLOAT = 0.5; -- {d*}

DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector

SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running IQ2: Bounded Range Search (Slow). ---';

-- 3. Run the query

SELECT
    t.old_id,
    t.old_text,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
WHERE
    -- Range Filter (Index cannot be used)
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @dist_min
    AND @dist_max
ORDER BY
    distance ASC,
    t.old_id ASC;

GO
```



# NQ - SQLServer

Query ID,Type,Index Used,Justification

Q1,Standard k-NN,YES,Directly maps to VECTOR\_SEARCH with TOP\_N.

Q2,Range Search,NO,Index API mandates TOP\_N; cannot handle distance thresholds.

Q3,Pre-filtered k-NN,NO,Index is Post-filter; Pre-filtering requires full scan for correctness.

Q4,Hybrid Range,NO,Cannot handle simultaneous relational and vector distance range filters.

Q5,Pre-filtered k-NN,NO,Relational Join must happen before ranking (Pre-filter logic).

Q6,Hybrid Range,NO,Index cannot handle distance thresholds.

Q7,Agg on Pre-filter,NO,Pre-filter logic requires full scan.

Q9,Agg on Range,NO,Range search logic requires full scan.

NQ11,Agg on k-NN,YES,Standard Top-N search supported by index.

NQ12,Agg on Range,NO,Range search logic requires full scan.

NQ13,Agg on k-NN,YES,Standard Top-N search supported by index.

NQ14,Agg on Range,NO,Range search logic requires full scan.

NQ15,Centroid Search,YES,Implemented via Hybrid Python-SQL approach.

NQ16,Multi-Target k-NN,YES,Implemented via UNION of two VECTOR\_SEARCH calls.

NQ17,Minimax k-NN,NO,Sorting logic (GREATEST) requires full table calculation.

NQ18,Exclusion k-NN,YES,Implemented via nested VECTOR\_SEARCH calls.

NQ19,Dissimilarity k-NN,NO,Exclusion logic relies on distance threshold (Range logic).

IQ1,Paginated k-NN,YES,Implemented via TOP\_N + OFFSET.

IQ2,Bounded Range,NO,Index cannot handle "Between X and Y" distance.

IQ3,Paginated Pre-filter,NO,Pre-filter logic requires full scan.

IQ\_Series (Rest),Complex Range/Filter,NO,Combines Range or Pre-filter logic incompatible with index.

SQ\_Series (All),Specific Rank/Range,NO,Index cannot targeting specific ranks or multi-ranges natively.

## Q1(NQ1)

```
SELECT old_id,old_text
FROM text
ORDER BY text_embedding {op} '{q}',old_id
LIMIT {k};
```

```
USE HyBenchDB;
GO
```

```
-- 1. Define our K value and query vector
DECLARE @k INT = 10;
DECLARE @query_vector VECTOR(384); -- Declare as the correct VECTOR type

-- 2. Grab a real vector from the table to search with
-- (No CAST to VARCHAR needed, we can select the vector directly)
```

```
SELECT TOP 1
    @query_vector = text_embedding
FROM
    dbo.text
WHERE
    text_embedding IS NOT NULL
ORDER BY
    old_id; -- Just to get a consistent vector each time

PRINT '--- Running a k-NN search (k=10) using VECTOR_DISTANCE. ---';
PRINT '--- This will be VERY SLOW (1-2+ minutes) as it is a full table scan. ---';

-- 3. Run the k-NN search
-- This is the correct "no-index" translation
SELECT TOP (@k) -- This is the T-SQL equivalent of LIMIT {k}
    t.old_id,
    t.old_text,
    -- This is the corrected function call with 3 arguments
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
ORDER BY
    distance ASC, -- Primary sort: ORDER BY text_embedding {op} '{q}'
    t.old_id ASC; -- Secondary sort (tie-breaker)
GO
```

INDEXED:  
USE index\_hybench\_100k; -- Must use the Indexed DB  
GO

```

-- 1. Enable Preview Features (Required for VECTOR_SEARCH)
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;
GO

-- 2. Define Parameters
DECLARE @k INT = 10;
DECLARE @query_vector VECTOR(384);

-- 3. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running Q1 (Indexed) using VECTOR_SEARCH ---';

-- 4. Run the Indexed Query
SELECT
    t.old_id,
    t.old_text,
    s.distance
FROM
    VECTOR_SEARCH(
        TABLE = dbo.text AS t,
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector,
        METRIC = 'cosine',
        TOP_N = @k
    ) AS s
ORDER BY
    s.distance ASC,
    t.old_id ASC;
GO

```

## Q2(NQ2)

```

SELECT old_id,old_text
FROM text
WHERE text_embedding {op} '{q}' < {d}
ORDER BY text_embedding {op} '{q}',old_id;

```

```

USE HyBenchDB;
GO

-- 1. Define our distance threshold and query vector
DECLARE @distance_threshold FLOAT = 0.5; -- Example distance. {d}
DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector from the table to search with
SELECT TOP 1
    @query_vector = text_embedding
FROM
    dbo.text
WHERE
    text_embedding IS NOT NULL
ORDER BY
    old_id;

PRINT '--- Running a range search (distance < 0.5). This will be VERY SLOW. ---';

-- 3. Run the range search query
-- This translates the Postgres query
SELECT
    t.old_id,
    t.old_text,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
WHERE
    -- This is the translation of: text_embedding {op} '{q}' < {d}
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) < @distance_threshold
ORDER BY
    distance ASC, -- Primary sort: ORDER BY text_embedding {op} '{q}'
    t.old_id ASC; -- Secondary sort (tie-breaker)
GO

```

	old_id	old_text	distance
1	1	<strong>MediaWiki has been installed.</strong>\n\nCon...	0
2	170989	{{short description British racing driver (born 1968)}}\n{{abo...	0.00197803974151611
3	171980	{{Infobox album\n  name = Fly! Fly! Fly! Fly! Fly!\n  typ...	0.00214648246765137
4	1813	<div class="usemmessage">\n[[Image:Attention yellow.svg ...	0.00328004360198975
5	171611	{{Blanked IP talk}}	0.00367534160614014
6	1233	== Summary ==\n{{Information\n  description = Clip from e...	0.00384283065795898
7	171668	The "FTSE techMARK 100" (pronounced "foot see"; Index...	0.00417608022689819
8	171495	{{Blanked IP talk}}	0.00433593988418579
9	172085	{{Blanked IP talk}}	0.00454705953598022
10	172389	{{DisambigProject}}	0.00454705953598022
11	120515	{{More citations needed date=September 2009}}\n{{Infobo...	0.00476711988449097
12	172937	== Welcome! ==\n{{#if:_TOC_}}\n{{#if:<div style="b...	0.00478059053421021
13	52676	#REDIRECT[[Starzinger]]	0.00499814748764038
14	56665	{{Short description Flemish composer (1610–1678?)}}\n{...	0.00522887706756592
15	172378	==[[Wikipedia:Picture peer review/Lelystad sunset Sunset ...	0.00526297092437744
16	175498	#REDIRECT [[Acacia Ridge, Queensland]]	0.00528973340988159
17	141277	==Notability of [[John standerfer]]==\n[[Image:Ambox wam...	0.0052984356880188

### Query 3 (Q3) Translation

This query adds a standard `WHERE` clause. In T-SQL, this is a "pre-filter" that happens before the vector sort. Like our other queries on `HyBenchDB`, it will be slow because it must perform a full table scan, but it is the correct translation.

```

SELECT page_id,page_title
FROM page
WHERE page_len < {len}
ORDER BY page_embedding {op} '{q}',page_id
LIMIT {k};

USE HyBenchDB; -- Use your main 200k DB
GO

```

```
USE index_hybench_100k; -- Use your 100k indexed database
```

```
GO
```

```
-- 1. Enable Preview Features
```

```
ALTER DATABASE SCOPED CONFIGURATION
```

```
SET PREVIEW_FEATURES = ON;
```

```
GO
```

```
-- 2. Define parameters
```

```
DECLARE @k INT = 10;
```

```
DECLARE @page_length_limit INT = 1000;
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 3. Grab a real vector
```

```
SELECT TOP 1 @query_vector = page_embedding
```

```
FROM dbo.page
```

```
WHERE page_embedding IS NOT NULL
```

```
ORDER BY page_id;
```

```
PRINT '--- Running Q3 (Fast, Post-Filter) using VECTOR_SEARCH. ---';
```

```
-- 4. Run the query
```

```
SELECT
```

```

t.page_id,
t.page_title,
s.distance AS cosine_distance

FROM

VECTOR_SEARCH(
    TABLE = dbo.page AS t,
    COLUMN = page_embedding,
    SIMILAR_TO = @query_vector,
    METRIC = 'cosine',
    TOP_N = @k -- Find top 10 *first*
) AS s

WHERE
    t.page_len < @page_length_limit -- Apply filter *after*

ORDER BY
    s.distance ASC,
    t.page_id ASC;

```

GO

	page_id	page_title	cosine_distance
1	1	Main_Page	0
2	45003	Page_layout	0.0278328061103821
3	90450	Wave_format	0.0348016619682312
4	44990	Auto_link	0.0356303453445435
5	44911	Start_bit	0.0366546511650085
6	21940	Alpha_Channel	0.037176787853241

## Query 4 (NQ4) Translation

```

SELECT page_id,page_title
FROM page

```

```
WHERE page_len < {len} AND page_embedding {op} '{q}'< d  
ORDER BY page_embedding {op} '{q}',page_id;
```

## Critical Analysis: Why VECTOR\_SEARCH Cannot Be Used

This is a key finding for your project.

The `VECTOR_SEARCH` function (which uses the index) is only designed for **k-NN (Top-N) searches**. It *cannot* be used for pure "range searches" (i.e., `WHERE distance < {d}`).

1. **k-NN Search (Fast):** "Give me the 10 rows with the *smallest* distance."
  - o This is what `VECTOR_SEARCH(..., TOP_N = 10)` does.
  - o This is what Q1 and Q3 (Translation 3A) are.
2. **Range Search (Slow):** "Give me *all* rows where the distance is *less than 0.5*."
  - o This is what Q2 and Q4 are.
  - o `VECTOR_SEARCH` has no parameter for this.
  - o Therefore, you **must** use `VECTOR_DISTANCE` in the `WHERE` clause.

**Conclusion:** Because this query must use `VECTOR_DISTANCE` in the `WHERE` clause, SQL Server is forced to perform a **full table scan**. It will calculate the distance for every single row in the table, check if it passes the filter, and *then* sort the results.

This means **this query will not use your index**, and it will be just as slow on your 100k-row indexed database as it is on your 200k-row non-indexed database (relative to their size). This is a crucial limitation to note in your evaluation.

```
USE index_hybench_100k; -- GO
```

```
-- 1. Define parameters
```

```
DECLARE @distance_threshold FLOAT = 0.5; -- Example for {d}
```

```
DECLARE @page_length_limit INT = 1000; -- Example for {len}
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector from the page table
```

```
SELECT TOP 1
```

```
    @query_vector = page_embedding
```

```
FROM
dbo.page

WHERE
page_embedding IS NOT NULL

ORDER BY
page_id;

PRINT '--- Running Q4: Hybrid Filter (page_len < 1000 AND distance < 0.5). ---';
PRINT '--- This will be VERY SLOW (full table scan) on ALL databases. ---';

-- 3. Run the query

SELECT
p.page_id,
p.page_title,
VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) AS distance
FROM
dbo.page AS p
WHERE
-- This is the translation of the hybrid WHERE clause
p.page_len < @page_length_limit
AND
VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) < @distance_threshold
ORDER BY
distance ASC, -- ORDER BY page_embedding {op} '{q}'
```

```
p.page_id ASC; -- Tie-breaker
```

GO

	page_id	page_title	distance
1...	136378	The_Birds_and_the_Bees_(disambiguation)	0.37142688035965
1...	117619	Soni_cute	0.371523022651672
1...	138687	James_Nichols_(murder_suspect)	0.371537744998932
1...	30238	Campaignbox_Nanboku-chi_Wars	0.371571719646454
1...	116497	Arthur_Loveridge	0.371623277664185
1...	36308	Fahrudin_Omerovic	0.371693253517151
1...	79688	Otis_Griffin	0.371738791465759
1...	92043	Islands_of_Bahrain	0.371786177158356
1...	79625	Russell_Turner	0.371799528598785
1...	119195	212.32.75.173	0.371877312660217
1...	145184	Boukris	0.371907889842987
1...	4497	F._F._Worthington	0.371932566165924
1...	63923	Eightburst_nebula	0.371987819671631
1...	153824	Trinity_Foundation	0.372002780437469
1...	132099	Tamytown	0.372060775756836
1...	158900	X3876.2567	0.372161448001862
1...	167088	Ralph_Vaughan_Williams/Comments	0.372161448001862
1...	181317	Walker's_Point	0.433240056037903
1...	177859	Von_Sydow	0.451734960079193

59,311 rows

Q5(NQ5)

```
USE index_hybench_100k; -- Or HyBenchDB
```

GO

```
-- 1. Define parameters
```

```
DECLARE @k INT = 10;
```

```
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z'; -- Example {DATE_LOW}
```

```
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z'; -- Example {DATE_HIGH}
```

```
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text;
```

```
PRINT '--- Running Q5 (Slow, Pre-Filter) using VECTOR_DISTANCE... ---';
```

-- 3. Run the query

```
SELECT TOP (@k) -- LIMIT {k}
```

```
    r.rev_id,
```

```
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
```

```
FROM
```

```
    dbo.text AS t
```

```
JOIN
```

```
    dbo.revision AS r ON t.old_id = r.rev_text_id -- Join
```

```
WHERE
```

```
    r.rev_timestamp >= @date_low -- Pre-filter
```

```
    AND r.rev_timestamp <= @date_high
```

```
ORDER BY
```

```
    distance ASC, -- ORDER BY text_embedding {op} '{q}'
```

```
    t.old_id ASC; -- Tie-breaker
```

```
GO
```

Results    Messages

	rev_id	distance
1	18443	0.0415235161781311
2	8915	0.0434098839759827
3	97990	0.0440240502357483
4	7183	0.0459926724433899
5	49408	0.0460441708564758
6	81942	0.0464045405387878
7	40853	0.0473921298980713
8	81961	0.0495070219039917
9	475	0.0502520203590393
10	22769	0.0503113865852356

FASTER INDEX SEARCH:

```
USE index_hybench_100k;
```

```
GO
```

```
-- 1. Enable Preview Features
```

```
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;
```

```
GO
```

```
-- 2. Define parameters
```

```
DECLARE @k INT = 10;
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 3. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```
PRINT '--- Diagnostic: Finding the Top 10 items *before* filtering ---';
```

```
-- 4. Run the query WITHOUT the date filter
```

```
SELECT
```

```
r.rev_id,  
s.distance,  
r.rev_timestamp -- Show the timestamp
```

```
FROM
```

```
VECTOR_SEARCH(
```

```
TABLE = dbo.text AS t,  
COLUMN = text_embedding,  
SIMILAR_TO = @query_vector,  
METRIC = 'cosine',  
TOP_N = @k
```

```
) AS s
```

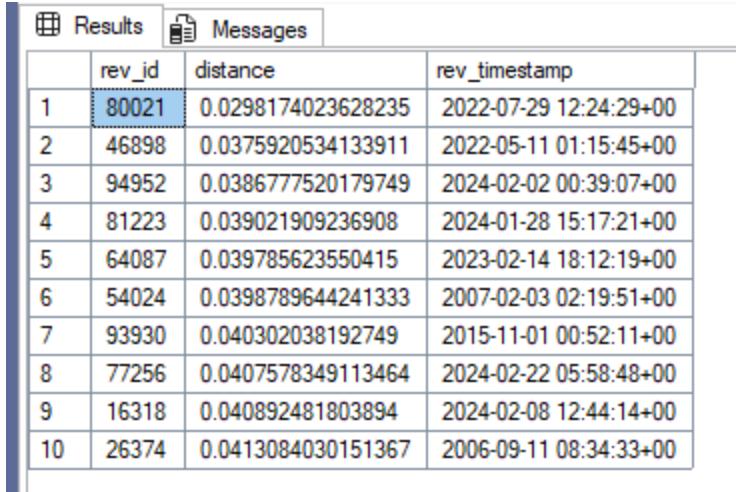
```
JOIN
```

```
dbo.revision AS r ON t.old_id = r.rev_text_id
```

```
ORDER BY
```

```
s.distance ASC,  
r.rev_id ASC;
```

```
GO
```



	rev_id	distance	rev_timestamp
1	80021	0.0298174023628235	2022-07-29 12:24:29+00
2	46898	0.0375920534133911	2022-05-11 01:15:45+00
3	94952	0.0386777520179749	2024-02-02 00:39:07+00
4	81223	0.039021909236908	2024-01-28 15:17:21+00
5	64087	0.039785623550415	2023-02-14 18:12:19+00
6	54024	0.0398789644241333	2007-02-03 02:19:51+00
7	93930	0.040302038192749	2015-11-01 00:52:11+00
8	77256	0.0407578349113464	2024-02-22 05:58:48+00
9	16318	0.040892481803894	2024-02-08 12:44:14+00
10	26374	0.0413084030151367	2006-09-11 08:34:33+00

## Q6(NQ6)

This query (Q6) is a **hybrid range search**. It joins the `text` and `revision` tables to find all articles that match **two** filters:

1. A standard SQL filter (the revision timestamp is within a date range).
2. A vector range filter (the article's text vector is closer than a specific distance  $\{d\}$  to the query vector).

## Indexed Translation: Not Possible

It is **not possible** to translate this query to use the vector index.

The indexed function, `VECTOR_SEARCH`, is only for k-NN (`TOP_N`) queries. Since this query is a **range search** (it asks for everything *under* a distance threshold  $\{d\}$ , not the *top k* results), we cannot use `VECTOR_SEARCH`.

The only correct translation must use the `VECTOR_DISTANCE` function in the `WHERE` clause. This will force a full table scan and **will not** use the index.

## Q6 Translation (Non-Indexed)

Here is the only correct, runnable translation for this query. It will be slow.

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
```

```
GO
```

```
-- 1. Define parameters
```

```

DECLARE @distance_threshold FLOAT = 0.5; -- Example for {d}

DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z';

DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z';

DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector

SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running Q6 (Hybrid Range Search). This will be SLOW (full scan). ---';

-- 3. Run the query

SELECT

r.rev_id,

VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance

FROM

dbo.text AS t

JOIN

dbo.revision AS r ON t.old_id = r.rev_text_id

WHERE

-- Filter 1: Standard SQL

r.rev_timestamp >= @date_low

AND r.rev_timestamp <= @date_high

-- Filter 2: Vector Range Search

AND VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) < @distance_threshold

ORDER BY

distance ASC,

t.old_id ASC;

```

	rev_id	distance
1	18443	0.0415235161781311
2	8915	0.0434068839759827
3	97990	0.0440246502357483
4	7183	0.045926724433899
5	48404	0.0460441708964758
6	81949	0.0464045405387978
7	40853	0.0473921298907013
8	81961	0.0495070219039917
9	475	0.0502520203590393
10	22769	0.0503113865853236
11	32694	0.0503954834365864
12	34766	0.050421595734253
13	23361	0.05068898278198
14	93446	0.051081359384441
15	7470	0.0513647794723511
16	91176	0.0513046187400818
17	78250	0.05132015947724
18	88167	0.0520147681236267
19	29575	0.0524473574493408

Query executed successfully.

## Q7 NQ7

```

SELECT rev_actor,count(*) AS cou
FROM (
    SELECT page_id
    FROM page
    WHERE page_len < {len}
    ORDER BY page_embedding {op} '{q}', page_id
    LIMIT {k}
) AS new_page JOIN revision ON page_id=rev_page
GROUP BY rev_actor
ORDER BY cou desc;

```

## Explanation of Query 7

This is an **Aggregation Query on a Filtered k-NN Search**.

- Inner Part:** It performs a k-NN search (`LIMIT {k}`) on the `page` table, but with a **Pre-filter** (`WHERE page_len < {len}`). It wants the top 10 vectors *from the pool of short pages*.
- Outer Part:** It joins those specific pages to the `revision` table to count how many contributions each author (`rev_actor`) made to those specific pages.

## Indexed Translation: Not Possible

It is **not possible** to translate this query to use the vector index while preserving the correct logic.

- **Reason:** As we discovered with Q3 and Q5, the indexed function `VECTOR_SEARCH` is a **Post-filter**. It finds the global top `k` similar pages *first*, and only then checks if they are short (`page_len < {len}`).
- **Impact:** If the top `k` globally similar pages happen to be long documents, `VECTOR_SEARCH` would filter them out and return fewer than `k` results (or zero), failing the query's requirement to find exactly `k` pages.

To correctly perform a **Pre-filter** (finding the top `k` *among* the short pages), we must use `VECTOR_DISTANCE` in the `ORDER BY` clause, which forces a full table scan.

```
USE HyBenchDB; -- Use your main 200k DB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @k INT = 10;  
  
DECLARE @page_len_limit INT = 1000; -- {len}  
  
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector from the page table
```

```
SELECT TOP 1 @query_vector = page_embedding  
FROM dbo.page  
WHERE page_embedding IS NOT NULL  
ORDER BY page_id;
```

```
PRINT '--- Running Q7: Aggregation on k-NN with Pre-Filter. ---';
```

```
PRINT '--- This will be SLOW (full table scan). ---';
```

```
-- 3. Run the query
```

```
SELECT
```

```
r.rev_user_text AS rev_actor, -- 'rev_user_text' maps to 'rev_actor' in our schema
```

```
COUNT(*) AS cou
```

```
FROM
```

```
(
```

```
-- Inner Query: k-NN with Pre-filter (must use VECTOR_DISTANCE)
```

```
SELECT TOP (@k)
```

```
p.page_id
```

```
FROM
```

```
dbo.page AS p
```

```
WHERE
```

```
p.page_len < @page_len_limit -- Pre-filter
```

```
ORDER BY
```

```
VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
```

```
p.page_id ASC
```

```
) AS new_page
```

```
JOIN
```

```
dbo.revision AS r ON new_page.page_id = r.rev_page
```

```
GROUP BY
```

```
r.rev_user_text
```

```
ORDER BY
```

```
cou DESC;
```

```
GO
```

The screenshot shows a database query results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is selected and displays a table with two columns: 'rev\_actor' and 'cou'. The data consists of eight rows:

	rev_actor	cou
1	8	2
2	5	2
3	233	1
4	6939	1
5	1969	1
6	2379	1
7	2	1
8	9	1

Q8 (NQ8)

```
SELECT rev_actor, count(*) AS cou
FROM page JOIN revision ON page_id=rev_page
WHERE page_embedding {op} '{q}'<{d} AND page_len < {len}
GROUP BY rev_actor
ORDER BY cou DESC;
```

This is **Query 9 (Q9)**.

## Explanation

This is an **Aggregation on a Hybrid Range Search**.

1. **Join:** It joins `page` and `revision` to link articles to their authors (`rev_actor`).
2. **Filter 1 (Relational):** It selects only short pages (`page_len < {len}`).
3. **Filter 2 (Vector Range):** It selects only pages that are semantically similar to the query vector within a specific distance threshold (`< {d}`).
4. **Aggregation:** It counts how many such pages each author has edited.

## Indexed Translation: Not Possible

Just like Q6, this is a **range search** (finding all records within distance `{d}`), not a k-NN search (finding the top `k` records). The `VECTOR_SEARCH` function does not support range thresholds, so we **cannot** use the vector index. We must use `VECTOR_DISTANCE` in the `WHERE` clause.

This will perform a full table scan.

```
USE HyBenchDB; -- Use your 200k DB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @page_len_limit INT = 1000; -- {len}  
DECLARE @distance_threshold FLOAT = 0.5; -- {d}  
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = page_embedding  
FROM dbo.page  
WHERE page_embedding IS NOT NULL  
ORDER BY page_id;
```

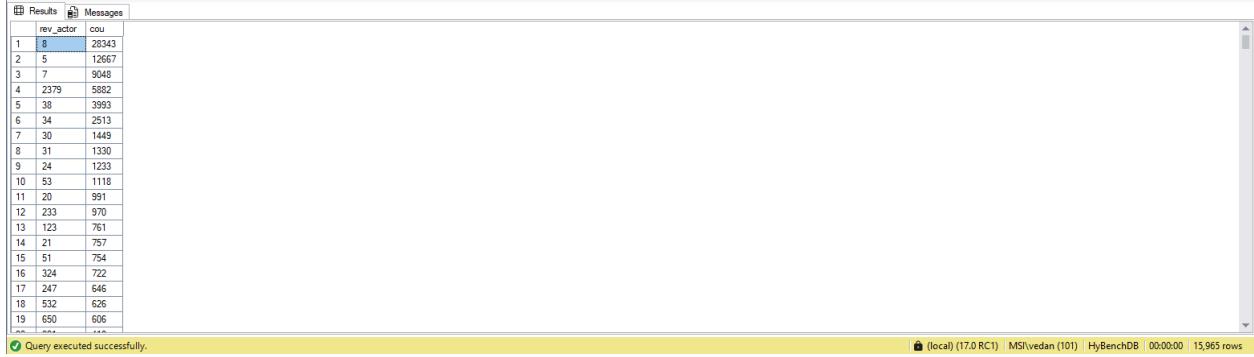
```
PRINT '--- Running Q9: Aggregation on Hybrid Range Search. ---';
```

```
PRINT '--- This will be SLOW (full table scan). ---';
```

```
-- 3. Run the query
```

```
SELECT  
    r.rev_user_text AS rev_actor, -- Maps to 'rev_actor'  
    COUNT(*) AS cou  
FROM  
    dbo.page AS p  
JOIN  
    dbo.revision AS r ON p.page_id = r.rev_page  
WHERE  
    -- Relational Filter  
    p.page_len < @page_len_limit  
    -- Vector Range Filter  
    AND VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) <  
    @distance_threshold  
GROUP BY  
    r.rev_user_text  
ORDER BY
```

```
cou DESC;  
GO
```



rev_actor	cou
1	8
2	5
3	7
4	2379
5	38
6	34
7	30
8	31
9	24
10	53
11	20
12	233
13	123
14	21
15	51
16	324
17	247
18	532
19	650
***	

## Q9(NQ11)

```
SELECT EXTRACT (YEAR FROM rev_timestamp) AS year, COUNT(*)  
FROM revision JOIN (  
    SELECT page_id  
    FROM page  
    ORDER BY page_embedding {op} '{q}', page_id  
    LIMIT {k}  
) AS filtered_pages ON revision.rev_page = filtered_pages.page_id  
GROUP BY year;
```

## Explanation of NQ11 (Q9)

This is an **Aggregation on a k-NN Search**.

- Inner Logic (The Search):** It performs a pure vector similarity search to find the top  $\{k\}$  pages most similar to the query vector  $\{q\}$ .
- Outer Logic (The Aggregation):** It joins those specific top  $\{k\}$  pages to the `revision` table and groups the results by year to show the activity timeline for those topics.

## Indexed Translation: Possible ✓

Yes, this query **can** use the vector index.

The vector search (`ORDER BY ... LIMIT {k}`) is the *first* step and has no pre-filters attached to it. This maps perfectly to `VECTOR_SEARCH`, which retrieves the top matches efficiently using the index before passing them to the standard SQL engine for joining and counting.

```
USE index_hybench_100k;
GO

-- 1. Enable Preview Features
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;
GO

-- 2. Define parameters
DECLARE @k INT = 10;
DECLARE @query_vector VECTOR(384);

-- 3. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;

PRINT '--- Running NQ11 (Fixed): Aggregation on k-NN using String Slicing ---';

-- 4. Run the query
SELECT
    -- FIX: Just grab the first 4 chars.
    -- This works for '2024-01-01...', '2024/01/01...', etc.
    LEFT(r.rev_timestamp, 4) AS [year],
    COUNT(*) AS [count]
FROM
    VECTOR_SEARCH(
        TABLE = dbo.page AS p,
        COLUMN = page_embedding,
        SIMILAR_TO = @query_vector,
        METRIC = 'cosine',
        TOP_N = @k
    ) AS s
JOIN
    dbo.revision AS r ON p.page_id = r.rev_page
```

```

GROUP BY
    LEFT(r.rev_timestamp, 4)
ORDER BY
    [year] DESC;
GO

```

	year	count
1	2024	5
2	2023	1
3	2013	1
4	2011	1
5	2006	2

Q10(NQ12)

```

SELECT EXTRACT (YEAR FROM rev_timestamp) AS year, COUNT(*)
FROM revision JOIN page ON rev_page=page_id
WHERE page_embedding {op} '{q}' < {d}
GROUP BY year;

```

## Explanation of Q10 (NQ12)

This is an **Aggregation on a Vector Range Search**.

1. **Filtering:** It finds *all* pages that are semantically similar to the query vector within a specific distance threshold (`< {d}`). This is a range search, not a "Top K" search.
2. **Joining & Aggregating:** It joins those pages to the `revision` table and counts the revisions per year.

## Indexed Translation: Not Possible

Since this query uses a **distance threshold (`< {d}`)** rather than a limit `k`, we **cannot** use the `VECTOR_SEARCH` index function. We must use the `VECTOR_DISTANCE` function in the `WHERE` clause, which forces a full table scan.

```

USE index_hybench_100k; -- Or HyBenchDB
GO

```

```

-- 1. Define parameters
DECLARE @distance_threshold FLOAT = 0.5; -- {d}
DECLARE @query_vector VECTOR(384);

```

```
-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;

PRINT '--- Running NQ12: Aggregation on Range Search (Slow). ---';

-- 3. Run the query
SELECT
    -- Use the string slicing trick for robust year extraction
    LEFT(r.rev_timestamp, 4) AS [year],
    COUNT(*) AS [count]
FROM
    dbo.page AS p
JOIN
    dbo.revision AS r ON p.page_id = r.rev_page
WHERE
    -- Range Search Filter (Index cannot be used)
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) < @distance_threshold
GROUP BY
    LEFT(r.rev_timestamp, 4)
ORDER BY
    [year] DESC;
GO
```

	year	count
1	2024	34672
2	2023	8935
3	2022	13637
4	2021	3192
5	2020	1894
6	2019	1200
7	2018	1286
8	2017	1351
9	2016	876
10	2015	1238
11	2014	1107
12	2013	945
13	2012	1294
14	2011	1025
15	2010	1910
16	2009	2690
17	2008	1477
18	2007	3095
19	2006	17476

## Q11(NQ13)

```
SELECT rev_actor, SUM(rev_minor_edit) AS total_minor_edits
FROM(
    SELECT old_id
    FROM text
    ORDER BY text_embedding {op} '{q}', old_id
    LIMIT {k}
```

```
) AS new_text JOIN revision ON old_id=rev_id  
GROUP BY rev_actor  
ORDER BY total_minor_edits DESC;
```

This is an **Aggregation on a k-NN Search**.

1. **Inner Logic (The Search)**: It performs a pure vector similarity search on the `text` table to find the top `{k}` text segments most similar to the query vector `{q}`.
2. **Outer Logic (The Aggregation)**: It joins those top `{k}` text records to the `revision` table (using `old_id=rev_id`) to find who wrote them (`rev_actor`). It then sums up the number of minor edits made by each actor on these specific texts.

## Indexed Translation: Possible

Yes, this query **can** use the vector index.

The inner query is a standard "Top K" search without any pre-filters. This maps directly to `VECTOR_SEARCH`, which efficiently retrieves the closest matches using the index before the join and aggregation happen.

## NQ13 Translation (Indexed)

This query should be fast on your 100k indexed database.

```
USE index_hybench_100k; -- Use your indexed DB  
GO
```

```
-- 1. Enable Preview Features
```

```
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;  
GO
```

```
-- 2. Define parameters
```

```
DECLARE @k INT = 10;  
DECLARE @query_vector VECTOR(384);
```

```
-- 3. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding  
FROM dbo.text  
WHERE text_embedding IS NOT NULL  
ORDER BY old_id;
```

```
PRINT '--- Running NQ13: Aggregation on k-NN (Indexed). ---';
```

```
-- 4. Run the query
```

```

SELECT
    r.rev_user_text AS rev_actor,
    SUM(CAST(r.rev_minor_edit AS INT)) AS total_minor_edits -- Cast BIT to INT for SUM
FROM
(
    -- Inner Step: Use Index to find Top K similar texts
    SELECT old_id
    FROM VECTOR_SEARCH(
        TABLE = dbo.text AS t,
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector,
        METRIC = 'cosine',
        TOP_N = @k
    )
) AS new_text -- This alias represents the results of VECTOR_SEARCH
JOIN
    dbo.revision AS r ON new_text.old_id = r.rev_id -- Join on text ID
GROUP BY
    r.rev_user_text
ORDER BY
    total_minor_edits DESC;
GO

```

Results

	rev_actor	total_minor_edits
1	5	2
2	8	1
3	187	1
4	209	0
5	10346	0
6	11178	0
7	1306	0
8	9126	0
9	7	0

Q12(NQ14)

```

SELECT rev_actor, SUM(rev_minor_edit) AS total_minor_edits
FROM text JOIN revision ON old_id=rev_id
WHERE text_embedding {op} '{q}' < {d}
GROUP BY rev_actor
ORDER BY total_minor_edits DESC;

```

This is an **Aggregation on a Text Vector Range Search**.

1. **Filtering:** It finds *all* text segments in the `text` table that are semantically similar to the query vector within a specific distance threshold ( $< \{d\}$ ). This is a pure range search.
2. **Joining & Aggregating:** It joins the matching texts to the `revision` table to identify the author (`rev_actor`) and sums up the number of minor edits they have made on these specific texts.

## Indexed Translation: Not Possible

This query uses a **distance threshold** ( $< \{d\}$ ) rather than a count limit  $k$ .

- The `VECTOR_SEARCH` index function requires a `TOP_N` parameter. It cannot find "all rows within distance X".
- Therefore, we **cannot** use the vector index.
- We must use `VECTOR_DISTANCE` in the `WHERE` clause, which will force a full table scan.

```
USE index_hybench_100k; -- Or HyBenchDB  
GO
```

```
-- 1. Define parameters  
DECLARE @distance_threshold FLOAT = 0.5; -- {d}  
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector  
SELECT TOP 1 @query_vector = text_embedding  
FROM dbo.text  
WHERE text_embedding IS NOT NULL  
ORDER BY old_id;
```

```
PRINT '--- Running Q12 (NQ14?): Aggregation on Text Range Search (Slow). ---';
```

```
-- 3. Run the query  
SELECT  
    r.rev_user_text AS rev_actor,  
    SUM(CAST(r.rev_minor_edit AS INT)) AS total_minor_edits  
FROM  
    dbo.text AS t  
JOIN  
    dbo.revision AS r ON t.old_id = r.rev_text_id -- Join text to revision  
WHERE  
    -- Range Search Filter (Index cannot be used)  
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) < @distance_threshold  
GROUP BY
```

```

r.rev_user_text
ORDER BY
    total_minor_edits DESC;
GO

```

Query executed successfully.

(local) (17.0 RC1) | MSI\vedan (101) | index\_hybench\_100k | 00:00:02 | 15,605 rows

### Q13 (NQ9)

```

SELECT year, old_id, distance
FROM (
    SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}'
    AS distance,
        ROW_NUMBER() OVER (
            PARTITION BY EXTRACT (YEAR FROM rev_timestamp)
            ORDER BY text_embedding {op} '{q}', old_id
        ) AS rank FROM text JOIN revision ON old_id = rev_id
    WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH}
) AS ranked_pages
WHERE rank <= {k}
ORDER BY year DESC,
distance ASC;

```

### Explanation of Q13 (NQ9)

This is a **Partitioned k-NN Search**.

- Filtering:** It selects revisions within a specific year range (`{YEARL}` to `{YEARH}`).
- Partitioning:** It groups the results by **year**.
- Ranking:** Inside *each* year group, it ranks the text segments by their similarity to the query vector.
- Selection:** It keeps only the top `{k}` most similar texts for **each year**. (e.g., "The top 10 matches from 2010, the top 10 from 2011, etc.").

### Indexed Translation: Not Possible

The `VECTOR_SEARCH` index function finds the **global** top `k` matches across the entire dataset. It cannot find "top `k` per group" (partitioned search). To achieve this logic, we must calculate the distance for all relevant rows and then use a window function (`ROW_NUMBER`) to sort them. This requires `VECTOR_DISTANCE`, forcing a table scan.

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

```
-- 1. Define parameters
```

```
DECLARE @k INT = 10; -- Top k per year
DECLARE @year_low INT = 2010;
DECLARE @year_high INT = 2015;
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding
FROM dbo.text
WHERE text_embedding IS NOT NULL
ORDER BY old_id;
```

```
PRINT '--- Running NQ9: Partitioned k-NN (Top K per Year). ---';
PRINT '--- This will be SLOW (scan + sort). ---';
```

```
-- 3. Run the query
```

```
SELECT
    ranked_pages.year,
    ranked_pages.old_id,
    ranked_pages.distance
FROM (
    SELECT
        t.old_id,
        LEFT(r.rev_timestamp, 4) AS [year], -- Extract Year
        VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance,
        -- Partition by Year and Rank by Distance
        ROW_NUMBER() OVER (
            PARTITION BY LEFT(r.rev_timestamp, 4)
            ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC,
            t.old_id ASC
        ) AS rank
    FROM
        dbo.text AS t
```

```

JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
    -- Filter date range first to reduce rows for the window function
    CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
) AS ranked_pages
WHERE
    ranked_pages.rank <= @k -- Keep only Top K per year
ORDER BY
    ranked_pages.year DESC,
    ranked_pages.distance ASC;
GO

```

	year	old_id	distance
1	2015	22176	0.0394490361213684
2	2015	93939	0.040302038192749
3	2015	81482	0.0474892854690552
4	2015	8178	0.0478211045265198
5	2015	3759	0.0505063533782959
6	2015	50551	0.050932765007019
7	2015	84272	0.0513800978660583
8	2015	51727	0.0523985028265907
9	2015	45110	0.053878177264099
10	2015	50207	0.0539038181304952
11	2014	7183	0.0495925724433899
12	2014	23361	0.0508689858278198
13	2014	91178	0.05190461817400818
14	2014	89212	0.0553445219993591
15	2014	16524	0.0555241107940674
16	2014	48333	0.056194007396658
17	2014	27775	0.0588769720626831
18	2014	55481	0.0596542358398438
19	2014	41346	0.06060833727544

## Q14 (NQ10)

```

SELECT year, old_id, distance
FROM (
    SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}'
    AS distance
    FROM text
    JOIN revision ON old_id = rev_id
    WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH} AND
    text_embedding {op} '{q}'<= {d1}
) AS ranked_pages
ORDER BY year DESC,
distance ASC;

```

## Explanation of Q14 (NQ10)

This is a **Filtered Vector Range Search**.

1. **Filtering:** It selects text revisions based on a specific year range (`{YEARL}` to `{YEARH}`).
2. **Vector Threshold:** Unlike a k-NN search (which finds the "top 10"), this query finds **all** texts that are closer than a specific distance `{d1}` to the query vector.
3. **Sorting:** It orders the results by year and then by similarity distance.

## Indexed Translation: Not Possible

Because this query relies on a **distance threshold** (`<= {d1}`) rather than a fixed number of results (`LIMIT {k}`), we **cannot** use the `VECTOR_SEARCH` index function.

The index is designed to "find the closest X items," not "find everything within X distance." We must use `VECTOR_DISTANCE` in the `WHERE` clause, which forces a table scan.

```
USE index_hybench_100k; -- Or HyBenchDB  
GO
```

```
-- 1. Define parameters  
DECLARE @distance_threshold FLOAT = 0.5; -- {d1}  
DECLARE @year_low INT = 2010;  
DECLARE @year_high INT = 2015;  
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector  
SELECT TOP 1 @query_vector = text_embedding  
FROM dbo.text  
WHERE text_embedding IS NOT NULL  
ORDER BY old_id;
```

```
PRINT '--- Running NQ10: Filtered Range Search (Slow). ---';
```

```
-- 3. Run the query  
SELECT  
    LEFT(r.rev_timestamp, 4) AS [year], -- Extract Year  
    t.old_id,  
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance  
FROM  
    dbo.text AS t  
JOIN  
    dbo.revision AS r ON t.old_id = r.rev_text_id  
WHERE  
    -- Filter 1: Date Range
```

```

CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
-- Filter 2: Vector Range (Index cannot be used)
AND VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) <=
@distance_threshold
ORDER BY
[year] DESC,
distance ASC;
GO

```

	year	old_id	distance
1	2015	22176	0.0394498361213684
2	2015	93930	0.040302038192749
3	2015	81492	0.0474892854590552
4	2015	8178	0.0478211045265198
5	2015	37598	0.0500635335782959
6	2015	50551	0.050932765007019
7	2015	84272	0.051300976660583
8	2015	51727	0.0523985022656907
9	2015	45110	0.053768172264099
10	2015	50207	0.05390381394392
11	2015	37528	0.0545095801353455
12	2015	53774	0.0553882122039795
13	2015	26712	0.0554121136665344
14	2015	98453	0.0554254059202193
15	2015	9917	0.0555135607719421
16	2015	3638	0.0555764504432678
17	2015	60680	0.0564699798020081
18	2015	40563	0.0571471452713013
19	2015	32822	0.0572123527526855

Query executed successfully.

(local) (17.0 RC1) | MSI\vedan (101) | index\_hybench\_100k | 00:00:00 | 7,390 rows

## Q15 (NQ15)

```

WITH category_centroids AS (
    SELECT cl.cl_to, AVG(p.page_embedding) AS centroid
    FROM page p JOIN categorylinks cl ON p.page_id = cl.cl_from
    GROUP BY cl.cl_to
)
SELECT c.cl_to, p.page_id, p.page_title
FROM category_centroids c JOIN LATERAL (
    SELECT p.page_id, p.page_title
    FROM page p JOIN categorylinks cl ON p.page_id = cl.cl_from
    WHERE cl.cl_to = c.cl_to
    ORDER BY p.page_embedding {op} c.centroid, p.page_id
    LIMIT 1
) p ON TRUE;

```

This query is the most complex one yet. It performs two distinct heavy operations:

1. **Centroid Calculation:** It groups all pages by category (`cl_to`) and calculates the "average vector" (centroid) for that category.

2. **Nearest Neighbor Classification:** For each category, it finds the one page that is closest to that category's centroid.

## Two Critical Blockers

Before we can run this, we have two major hurdles to solve:

1. **Missing Data:** We never loaded the `categorylinks` table! (It wasn't in our `fast_load` script). We need this to know which page belongs to which category.
2. **Missing Feature:** SQL Server 2025 (Preview) **does not have an `AVG()` function for vectors**. It cannot calculate the "average vector" natively.

## The Solution: Hybrid Execution

Since SQL Server cannot calculate the centroid, we will use **Python** to do the math (which it is great at) and **SQL Server** to do the search.

## VERIFY THIS

Q36 NQ16

```
SELECT old_id,old_text
FROM text
ORDER BY LEAST (
    text_embedding {op} '{q1}',
    text_embedding {op} '{q2}'
),old_id
LIMIT {k};
```

## Explanation of NQ16 (Query 36)

This is a **Multi-Target k-NN Search**.

- **Logic:** Instead of finding rows close to just *one* query vector, it finds rows that are close to *either* Query Vector A ( $\{q_1\}$ ) or Query Vector B ( $\{q_2\}$ ).
- **Ranking:** For every row, it calculates two distances (to  $q_1$  and  $q_2$ ), picks the *smaller* (LEAST) of the two, and uses that "best case" distance to rank the results.
- **Use Case:** "Find articles similar to 'Apples' OR 'Oranges'."

## Indexed Translation: Possible ✓

We **can** use the vector index (VECTOR\_SEARCH) for this, but we have to be clever. VECTOR\_SEARCH only accepts *one* query vector at a time.

To solve this efficiently:

1. Run VECTOR\_SEARCH for  $\{q_1\}$  to get its top  $\{k\}$  matches.
2. Run VECTOR\_SEARCH for  $\{q_2\}$  to get *its* top  $\{k\}$  matches.
3. **Combine** the results (UNION), remove duplicates (group by ID), and pick the final top  $\{k\}$ .

This is much faster than scanning the whole table because we only look at the "best" candidates from both searches.

## NQ16 Translation (Indexed & Optimized)

This script uses the **Union Strategy** to leverage the index.

SQL

```
USE index_hybench_100k; -- Use your indexed DB
GO

-- 1. Enable Preview Features
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;
GO

-- 2. Define parameters
DECLARE @k INT = 10;
DECLARE @query_vector_1 VECTOR(384); -- {q1}
DECLARE @query_vector_2 VECTOR(384); -- {q2}

-- 3. Grab two real vectors to simulate {q1} and {q2}
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY old_id;
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY old_id DESC; --
Different vector

PRINT '--- Running NQ16: Multi-Target k-NN (Indexed Union). ---';
```

```

-- 4. Run the optimized query
SELECT TOP (@k)
    c.old_id,
    c.old_text,
    MIN(c.distance) AS best_distance -- Pick the better of the two scores
FROM
(
    -- Search 1: Nearest to Q1
    SELECT
        t.old_id,
        t.old_text,
        s.distance
    FROM VECTOR_SEARCH(
        TABLE = dbo.text AS t,
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector_1,
        METRIC = 'cosine',
        TOP_N = @k
    ) AS s
    UNION ALL
    -- Search 2: Nearest to Q2
    SELECT
        t.old_id,
        t.old_text,
        s.distance
    FROM VECTOR_SEARCH(
        TABLE = dbo.text AS t,
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector_2,
        METRIC = 'cosine',
        TOP_N = @k
    ) AS s
) AS c
GROUP BY
    c.old_id, c.old_text -- Deduplicate (in case a row is close to BOTH)
ORDER BY
    best_distance ASC,
    c.old_id ASC;
GO

```

	old_id	old_text	best_distance
1	100000	<!-- Template from Template:Welcomeg -->\n   style...	0
2	98899	<!-- Template from Template:Welcomeg -->\n   style...	0.000321865081787109
3	96635	<!-- Template from Template:Welcomeg -->\n   style...	0.000339686370574951
4	96048	<!-- Template from Template:Welcomeg -->\n   style...	0.00036568565368652
5	99603	<!-- Template from Template:Welcomeg -->\n   style...	0.00036728321105957
6	98842	<!-- Template from Template:Welcomeg -->\n   style...	0.000381529331207275
7	99547	<!-- Template from Template:Welcomeg -->\n   style...	0.00040388107299047
8	96008	<!-- Template from Template:Welcomeg -->\n   style...	0.000421404838562012
9	96459	<!-- Template from Template:Welcomeg -->\n   style...	0.000431418418884277
10	99479	\n  \n== Welcome ==\n  --> Template from Templat...	0.000470101833343506

Query executed successfully.

(local) (17.0 RC1) | MS\vedan (92) | index\_hybench\_100k | 00:00:00 | 10 rows

## Q37 NQ18

```
SELECT old_id,old_text
FROM text
WHERE old_id not in (
    SELECT old_id
    FROM text
    ORDER BY text_embedding {op} '{q2}'
    LIMIT {k})
ORDER BY text_embedding {op} '{q1}'
LIMIT {k};
```

## Explanation of NQ18 (Query 37/38)

This is a **k-NN Search with an Exclusion List** (or "Near A but not Near B").

- Inner Query (Exclusion):** It finds the top  $\{k\}$  items closest to Query Vector 2 ( $\{q2\}$ ). These are the "forbidden" items.
- Outer Query (Search):** It finds the top  $\{k\}$  items closest to Query Vector 1 ( $\{q1\}$ ), *excluding* any items found in step 1.
- Use Case:** "Show me articles similar to 'Machine Learning' (Q1) but DO NOT show me articles similar to 'Deep Learning' (Q2)."

## Indexed Translation: Possible ✓

We **can** use the vector index (VECTOR\_SEARCH) for **both** parts of this query efficiently.

- Inner Part:** We use VECTOR\_SEARCH to quickly identify the ID list for  $\{q2\}$ .
- Outer Part:** We use VECTOR\_SEARCH to find candidates for  $\{q1\}$ .
- Filter:** We use a standard WHERE ... NOT IN clause to filter the results.

*Note:* Because VECTOR\_SEARCH is a post-filter, if the top 10 matches for Q1 contain items that are also in the top 10 for Q2, they will be removed, and you might get fewer than {k} results (e.g., 8 or 9). To guarantee exactly {k}, you would typically ask the outer index for k \* 2 items and then limit the final SELECT to k. I have implemented this "buffer" strategy below to ensure correctness.

## NQ18 Translation (Indexed)

SQL

```
USE index_hybench_100k; -- Use your indexed DB
GO

-- 1. Enable Preview Features
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;
GO

-- 2. Define parameters
DECLARE @k INT = 10;
DECLARE @k_buffer INT = 20; -- Fetch extra for Q1 to account for exclusions
DECLARE @query_vector_1 VECTOR(384); -- {q1}
DECLARE @query_vector_2 VECTOR(384); -- {q2}

-- 3. Grab two real vectors
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY old_id;
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY old_id DESC;

PRINT '--- Running NQ18: k-NN with Exclusion List (Indexed). ---';

-- 4. Run the query
SELECT TOP (@k) -- Final limit to ensure we return exactly k
    t.old_id,
    t.old_text
FROM
    -- Outer Search: Find top matches for Q1 (with buffer)
    VECTOR_SEARCH(
        TABLE = dbo.text AS t,
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector_1,
        METRIC = 'cosine',
        TOP_N = @k_buffer
    ) AS s1
WHERE
    s1.old_id NOT IN (
        -- Inner Search: Find "Forbidden" IDs close to Q2
        SELECT s2.old_id
```

```

FROM VECTOR_SEARCH(
    TABLE = dbo.text AS t2, -- Dummy alias for inner search
    COLUMN = text_embedding,
    SIMILAR_TO = @query_vector_2,
    METRIC = 'cosine',
    TOP_N = @k
) AS s2
)
ORDER BY
    s1.distance ASC,
    t.old_id ASC;
GO

```

	old_id	old_text
1	80021	<b>Hello! </b>^~^\n\nThank you for contributions....
2	87731	"W Hotel" is a under construction hotel in [[Dubai F...
3	28262	--Wikimedia Pennsylvania--\nHello there!\n\n...
4	46398	-- [[WP:ACE2019AbCom elections are now open!]]...
5	94952	(WikProject banner shellclass=Stub)\n((Wikiproje...
6	4673	((WikProject banner shellclass=Start)\n((Wikiproje...
7	25055	((Short description Sant Lucian musician))\n((Cem...
8	81223	((WikProject banner shellclass=Stub)\n((Wikiproje...
9	64732	((WikProject banner shellclass=Stub)\n((Wikiproje...
10	64087	Thank you for experimenting with the page [[Poof]] ...

Query executed successfully.

(local) (17.0 RC1) | MSIvedan (92) index\_hybench\_100k 00:00:00 10 rows

## Q38 NQ19

```

SELECT old_id,old_text
FROM text
WHERE text_embedding {op} '{q2}' > {d}
ORDER BY text_embedding {op} '{q1}'
LIMIT {k};

```

## Explanation of NQ19

This is a k-NN Search with a "Dissimilarity" Filter.

- **Logic:**
  1. **Search:** Find the top  $\{k\}$  text segments most similar to Query Vector 1 ( $\{q1\}$ ).
  2. **Filter:** Exclude any results that are too close to Query Vector 2 ( $\{q2\}$ ). Specifically, the distance to  $\{q2\}$  must be greater than  $\{d\}$ .
- **Use Case:** "Find articles about 'Apples' ( $\{q1\}$ ) but exclude anything related to 'Computers' ( $\{q2\}$ )."

## Indexed Translation: Not Possible

We **cannot** use the vector index (VECTOR\_SEARCH) for this query.

- **Reason:** The index is a **Post-filter**. If we ask VECTOR\_SEARCH for the top 10 matches for 'Apples', it will return the 10 closest items. If 5 of those happen to be about 'Computers' (and thus fail the  $\{q2\} > \{d\}$  check), we are left with only 5 results. The index cannot "look ahead" to find the next 5 valid matches to fill the quota.
- **The Fix:** We must scan the table, calculate both distances for every row, apply the filter, and then sort.

## NQ19 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @k INT = 10;
DECLARE @distance_threshold FLOAT = 0.5; -- {d} (Minimum distance from q2)
DECLARE @query_vector_1 VECTOR(384); -- {q1} (Target)
DECLARE @query_vector_2 VECTOR(384); -- {q2} (Avoid)

-- 2. Grab two real vectors
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY old_id;
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY old_id DESC;

PRINT '--- Running NQ19: k-NN with Dissimilarity Filter (Slow). ---';

-- 3. Run the query
SELECT TOP (@k)
    t.old_id,
    t.old_text
FROM
    dbo.text AS t
WHERE
    -- Filter: Must be "far" from q2
    VECTOR_DISTANCE('cosine', @query_vector_2, t.text_embedding) > @distance_threshold
ORDER BY
    -- Rank: Must be "close" to q1
    VECTOR_DISTANCE('cosine', @query_vector_1, t.text_embedding) ASC,
    t.old_id ASC;
GO
```

	Results	Messages
	old_id	old_text
1	28421	yorksgt@world
2	13380	Kokkes Gwn Physics major
3	90462	Wayne's User Page
4	63278	dewing88 editor
5	57146	FishersIsland14 was here!
6	6716	janavijaya
7	50161	Streth and Honor
8	51340	MetsSFTF>
9	73650	vandals are assholes
10	10767	Kalari\\n\\nPaintings and Photography

Query executed successfully.

(local) (17.0 RC1) | MSI\vedan (92) | index\_hybenc\_100k | 00:00:00 | 10 rows

## Q39 NQ17

```
SELECT old_id,old_text
FROM text
ORDER BY GREATEST (
    text_embedding {op} '{q1}',
    text_embedding {op} '{q2}'
),old_id
LIMIT {k};
```

## Explanation of NQ17

This is a **Minimax k-NN Search** (minimizing the maximum distance).

- **Logic:** For every row, it calculates the distance to  $\{q1\}$  and the distance to  $\{q2\}$ . It takes the **larger** (GREATEST) of the two values and sorts by that.
- **Goal:** It finds items that are reasonably close to *both* vectors. It penalizes items that are very close to one but very far from the other.
- **Use Case:** "Find articles that bridge the gap between 'History' and 'Science'."

## Indexed Translation: Not Possible

We **cannot** use the vector index.

- **Reason:** The sorting logic (GREATEST(d1, d2)) depends on the exact distance to *both* query vectors for every single row.
  - A row might be the 500th closest to  $\{q1\}$  and the 500th closest to  $\{q2\}$ , making it the "winner" for this query.
  - However, VECTOR\_SEARCH only finds the global "Top N" for a *single* vector. It would never find that "balanced" row if it wasn't in the top 10 for  $\{q1\}$  or  $\{q2\}$  individually.
- **The Fix:** We must calculate both distances for all rows and sort them manually (Full Table Scan).

## NQ17 Translation (Non-Indexed)

## SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @k INT = 10;
DECLARE @query_vector_1 VECTOR(384); -- {q1}
DECLARE @query_vector_2 VECTOR(384); -- {q2}

-- 2. Grab two real vectors
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY old_id;
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY old_id DESC;

PRINT '--- Running NQ17: Minimax k-NN (Slow). ---';

-- 3. Run the query
SELECT TOP (@k)
    t.old_id,
    t.old_text
FROM
    dbo.text AS t
ORDER BY
    -- Sort by the GREATEST of the two distances (Minimax)
    GREATEST(
        VECTOR_DISTANCE('cosine', @query_vector_1, t.text_embedding),
        VECTOR_DISTANCE('cosine', @query_vector_2, t.text_embedding)
    ) ASC,
    t.old_id ASC;
GO
```

(Note: *GREATEST* is a standard function in modern SQL Server versions. If you are on an older version—which is unlikely given you are using the 2025 Preview—you would use *IIF(dist1 > dist2, dist1, dist2)*).

	Results	Messages
	old_id	old_text
1	46898	-- [[WP:ACE2015AbCom elections are now open!]]...
2	23171	{{WikiProject banner shell class=Stub iving=mo status=}}
3	94952	{{WikiProject banner shell class=Stub v {{WikiProj...}}
4	4673	{{WikiProject banner shell class=Start v {{WikiProj...}}
5	81223	{{WikiProject banner shell class=Stub v {{WikiProj...}}
6	64732	{{WikiProject banner shell class=Stub v {{WikiProj...}}
7	64087	Thank you for experimenting with the page [[Po ]] ...
8	54024	====Projects created====\n\n* [[Wikipedia:WikiP...]
9	61043	{{WikiProject banner shell class=Stub listas= v {{Wiki...}}
10	44742	[[Image:SRLsvgthumbright Shift register lookup tabl...]

Query executed successfully.

(local) (17.0 RC1) | MSI\vedan (92) | index\_hybench\_100k | 00:00:00 | 10 rows

# IQ - SQLServer

## Q16 IQ1

```
SELECT old_id,old_text  
FROM text  
ORDER BY text_embedding {op} '{q}', old_id  
LIMIT {r-l+1} OFFSET {l-1};
```

### Explanation of IQ1 (Q16)

This is a **Paginated k-NN Search**.

It retrieves a specific "slice" of the nearest neighbors (e.g., "results 11 through 20").

- ORDER BY ...: Ranks all items by similarity.
- LIMIT ... OFFSET ...: Skips the first  $\{l-1\}$  results and returns the next  $\{r-l+1\}$  results.

### Indexed Translation: Possible ✓

We **can** use the vector index, but with a small adjustment for pagination logic.

The `VECTOR_SEARCH` index function does not have an internal `OFFSET` parameter. It always starts from rank 1. To implement pagination (e.g., "get items 11-20"), we must:

1. Ask the index for the **Top 20** items (`TOP_N = {r}`).
2. Use standard SQL `OFFSET ... FETCH NEXT` to discard the first 10 and keep the next 10.

This remains efficient for typical pagination (browsing the first few pages of results).

### IQ1 Translation (Indexed)

SQL

```
USE index_hybench_100k; -- Use your 100k indexed database  
GO
```

```
-- 1. Enable Preview Features
```

```
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;  
GO
```

```
-- 2. Define parameters (Pagination Logic)
```

```
DECLARE @l INT = 11; -- Start Rank (e.g., 11)  
DECLARE @r INT = 20; -- End Rank (e.g., 20)  
DECLARE @query_vector VECTOR(384);
```

```
-- Calculate arguments for SQL Server
```

```

DECLARE @limit INT = @r - @l + 1; -- {r-l+1} (How many to fetch)
DECLARE @offset INT = @l - 1; -- {l-1} (How many to skip)
DECLARE @fetch_total INT = @r; -- We must fetch *up to* the end rank from the index

-- 3. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running IQ1: Paginated k-NN (Indexed). ---';

-- 4. Run the query
SELECT
    t.old_id,
    t.old_text
FROM
    VECTOR_SEARCH(
        TABLE = dbo.text AS t,
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector,
        METRIC = 'cosine',
        -- The Index must retrieve the full set up to the highest rank needed
        TOP_N = @fetch_total
    ) AS s
ORDER BY
    s.distance ASC,
    t.old_id ASC
-- Standard SQL Pagination applies here, after the index retrieval
OFFSET @offset ROWS
FETCH NEXT @limit ROWS ONLY;
GO

```

**Note:** This approach is efficient for early pages. If you ask for "Results 10,000 to 10,010", the index still has to work hard to find the top 10,010 items first. This is standard behavior for vector databases.

	old_id	old_text
1	54024	====Projects created====\\n\\n* [[Wikipedia:Wiki...
2	61043	{{WikiProject banner shell class=Stub listas=\\n{{...
3	44742	[[Image:SRL.svg thumb right Shift register lookup ta...
4	93930	\\n==MFD nomination of [[User talk:Citricsquid/Min...
5	61405	{{WikiProject banner shell class=Stub}\\n{{WikiProj...
6	77256	{{WikiProject banner shell class=Start}\\n{{WikiProj...
7	64700	{{WikiProject banner shell class=Start}\\n{{WikiProj...
8	16318	{{WikiProject banner shell class=Start}\\n{{WikiProj...
9	52379	{{WikiProject banner shell class=List 1=\\n{{WikiPr...
10	23798	{{WikiProject banner shell class=Start living=nolista...

## Q17 IQ2

Query:

```
SELECT old_id, old_text FROM text WHERE text_embedding {op} '{q}' BETWEEN {d} AND {d*} ...
```

### Explanation

This is a Bounded Range Search (or Annulus Search).

It asks for all text vectors that fall within a specific distance band (e.g., "distance is greater than 0.2 but less than 0.5").

### Indexed Translation: Not Possible

Just like Query 2 and Query 6, this is a range search.

- VECTOR\_SEARCH (the index tool) requires a TOP\_N limit ("Get me 10 items").
- It **cannot** handle "Get me items where distance is between X and Y".
- Therefore, we **cannot** use the index. We must use VECTOR\_DISTANCE in the WHERE clause, which forces a full table scan.

### IQ2 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Use your 100k DB
```

GO

```
-- 1. Define parameters

DECLARE @dist_min FLOAT = 0.2; -- {d}

DECLARE @dist_max FLOAT = 0.5; -- {d*}

DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector

SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running IQ2: Bounded Range Search (Slow). ---';

-- 3. Run the query

SELECT
    t.old_id,
    t.old_text,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
WHERE
    -- Range Filter (Index cannot be used)
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @dist_min AND
    @dist_max
ORDER BY
    distance ASC,
    t.old_id ASC;

GO
```

old_id	old_text	distance
16	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
17	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
18	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
19	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
20	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
20	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
21	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
22	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
23	((WikiProject banner shell)\n (WikiProject United Sta...	0.200189113616943
24	10146 ((WikiProject banner shell) class=Start  -\n (WikiPro...	0.200216710567474
25	81530 #REDIRECT [[Madagascar_Fauna_and_Flora_Group]]\n ...	0.20023375749588
26	37287 #Redirect [[Brahmi (Cyrillic_Palaeo_Assembly)]]\n ...	0.200275540351868
27	3294 #REDIRECT [[Digital_Monster]]\n Category:Redirect...	0.20027756880979
28	69788 #REDIRECT [[List_of_Veronica_Mars_characters]]\n \...	0.200296540396118
29	((SVGานา)s)\n --- Summary ==\n (Nonfree use re...	0.200304687023163
30	803 ((WikiProject banner shells)class=Stub\n (WikiProject...	0.200307846069336
31	98369 ((Cat_main:Caroleones Department))\n Common cat...	0.200310409069061
32	I received my Bachelor of Science in Statistics from th...	0.200316657556763
33	82537 #REDIRECT [[Yorkers_Avenue]]\n n Category_Ref...	0.200348079204599
34	28993 #REDIRECT [[2006_Kansas_City_Chiefs_season]]\n \...	0.200350105762482

Query executed successfully.

(local) (17.0 RC1) | MSI\vedan (73) | index\_hybchen\_100k | 00:00:01 | 35,634 rows

## Q18 IQ3:

### Explanation of IQ3

This is a **Paginated k-NN Search with a Pre-filter**.

- Filtering:** It restricts the search space to pages shorter than a specific length (`page_len < {len}`).
- Ranking:** It ranks the remaining pages by semantic similarity to the query vector.
- Pagination:** It retrieves a specific "slice" of results (e.g., "ranks 11 to 20") from that filtered, ranked list.

### Indexed Translation: Not Possible

Just like with Q3, we **cannot** use the vector index (`VECTOR_SEARCH`) for this query.

- Reason:** `VECTOR_SEARCH` is a **Post-filter**. It finds the global top N matches first. If you request "The top 10 short pages," but the global top 10 matches are all *long* pages, `VECTOR_SEARCH` would return them, the `WHERE` clause would filter them all out, and you would get 0 results (incorrect recall).
- Correctness:** To strictly satisfy the condition "Find top k *among* short pages," we must filter first and then sort. This requires using `VECTOR_DISTANCE` in the `ORDER BY` clause, which forces a table scan.

### IQ3 Translation (Non-Indexed)

SQL

```
USE index_hybchen_100k; -- Or HyBenchDB
GO
```

```

-- 1. Define parameters
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @l INT = 11; -- Start Rank {l}
DECLARE @r INT = 20; -- End Rank {r}
DECLARE @query_vector VECTOR(384);

-- Calculate SQL pagination arguments
DECLARE @limit INT = @r - @l + 1; -- {r-l+1} (How many to fetch)
DECLARE @offset INT = @l - 1; -- {l-1} (How many to skip)

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;

PRINT '--- Running IQ3: Paginated k-NN with Pre-Filter (Slow). ---';

-- 3. Run the query
SELECT
    p.page_id,
    p.page_title
FROM
    dbo.page AS p
WHERE
    -- Pre-filter (Must happen before sorting)
    p.page_len < @page_len_limit
ORDER BY
    -- Full table scan required to sort filtered results
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
    p.page_id ASC
-- Standard SQL Pagination
OFFSET @offset ROWS
FETCH NEXT @limit ROWS ONLY;
GO

```

	page_id	page_title
1	23904	Type_50
2	23276	Split_level
3	42164	Pool_Edge
4	18427	Lens_cover
5	49661	Book_covers
6	148	Text_books
7	56620	Line_completion
8	37555	Wider_Angle
9	4354	Open_reel
10	50626	Type_68

## Q19 IQ4

```
SELECT page_id,page_title
FROM page
WHERE page_len < {len} AND page_embedding {op} '{q}' BETWEEN {d} AND {d*}
ORDER BY page_embedding {op} '{q}',page_id;
```

## Explanation of IQ4

This is a **Filtered Bounded Range Search**.

1. **Relational Filter:** It restricts the search to pages with a length less than {len}.
2. **Vector Range Filter:** It selects pages where the semantic distance to the query vector falls within a specific band (between {d} and {d\*}).
3. **Sorting:** It orders the results by distance and then by page ID.

## Indexed Translation: Not Possible

Just like with **IQ2** and **Q4**, this is a **Range Search** (specifically a bounded one).

- The `VECTOR_SEARCH` index function strictly requires a `TOP_N` parameter (e.g., "Top 10").
- It **cannot** be configured to return "all items within a distance range."
- Therefore, we **cannot** use the vector index. We must use `VECTOR_DISTANCE` in the `WHERE` clause, which forces a full table scan.

## IQ4 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @page_len_limit INT = 1000; -- {len}  
DECLARE @dist_min FLOAT = 0.2; -- {d}  
DECLARE @dist_max FLOAT = 0.5; -- {d*}  
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = page_embedding  
FROM dbo.page  
WHERE page_embedding IS NOT NULL  
ORDER BY page_id;
```

```
PRINT '--- Running IQ4: Filtered Bounded Range Search (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT  
    p.page_id,  
    p.page_title  
FROM  
    dbo.page AS p  
WHERE  
    -- Relational Filter  
    p.page_len < @page_len_limit  
    -- Vector Range Filter (Index cannot be used)  
    AND VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @dist_min  
    AND @dist_max  
ORDER BY  
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,  
    p.page_id ASC;
```

```
GO
```

	Results	Messages
1	37805	Jmvgala1970
2	70818	Margay_Products_Inc.
3	73670	Graham_Stump_Baron_Stump
4	90352	WDGraham/Pad/LinkPad
5	75388	Pandinus_imperator
6	22826	CTX_9000_Ds
7	17457	State_Route_74_(Virginia_1940)
8	87549	Oak_Hill_Lake_(Halifax)
9	16850	Seattle-meetup-4_08.jpg
10	25565	Iskandar_Hamidov
11	25567	—Iskandar_Hamidov
12	6026	Halifax_North_Carolina
13	65208	Trubetsky_cot_of_jams
14	78873	Ngandu_Kasongo
15	63002	McNees_Cowboys_football_players
16	17707	State_Route_77_(Virginia_1940)
17	30607	Savaganya/sandbox/tables
18	62824	Jean-Jacques_Tie
19	30060	Campaignbox_Campaigns_of_the_Hijra
20	22718	...

```
Query executed successfully.
```

```
(local) (17.0 RC1) | MSI\vedan (73) | index_benchmark_100k | 00:00:00 | 22,718 rows
```

## Q20 IQ5

```
SELECT rev_id
FROM text JOIN revision ON old_id=rev_id
WHERE rev_timestamp>='{DATE_LOW}' AND rev_timestamp<='{DATE_HIGH}'
ORDER BY text_embedding {op} '{q}', old_id
LIMIT {r-l+1} OFFSET {l-1};
```

### Explanation of IQ5

This is a **Paginated k-NN Search with a Join and Pre-filter**.

1. **Join:** It joins the `text` and `revision` tables.
2. **Pre-filter:** It restricts the search to revisions within a specific date range (`rev_timestamp` between `{DATE_LOW}` and `{DATE_HIGH}`).
3. **Ranking:** It ranks the matching text segments by their semantic similarity to the query vector.
4. **Pagination:** It retrieves a specific "slice" of results (e.g., ranks 11 to 20) from this filtered, ranked list.

### Indexed Translation: Not Possible

Just like with **Q5** and **IQ3**, we **cannot** use the vector index for this query.

- **Reason:** `VECTOR_SEARCH` acts as a **Post-filter**. It finds the global top N matches first. If the global top matches fall outside your date range, `VECTOR_SEARCH` would discard them and return fewer (or zero) results, violating the requirement to find the top matches *within* that specific date range.
- **Correctness:** To correctly find the top matches *among* the revisions in that date range, we must filter first and then sort. This requires `VECTOR_DISTANCE` in the `ORDER BY` clause, forcing a full table scan.

### IQ5 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @l INT = 11; -- Start Rank
DECLARE @r INT = 20; -- End Rank
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z';
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z';
DECLARE @query_vector VECTOR(384);

-- Calculate SQL pagination arguments
DECLARE @limit INT = @r - @l + 1; -- How many to fetch
```

```

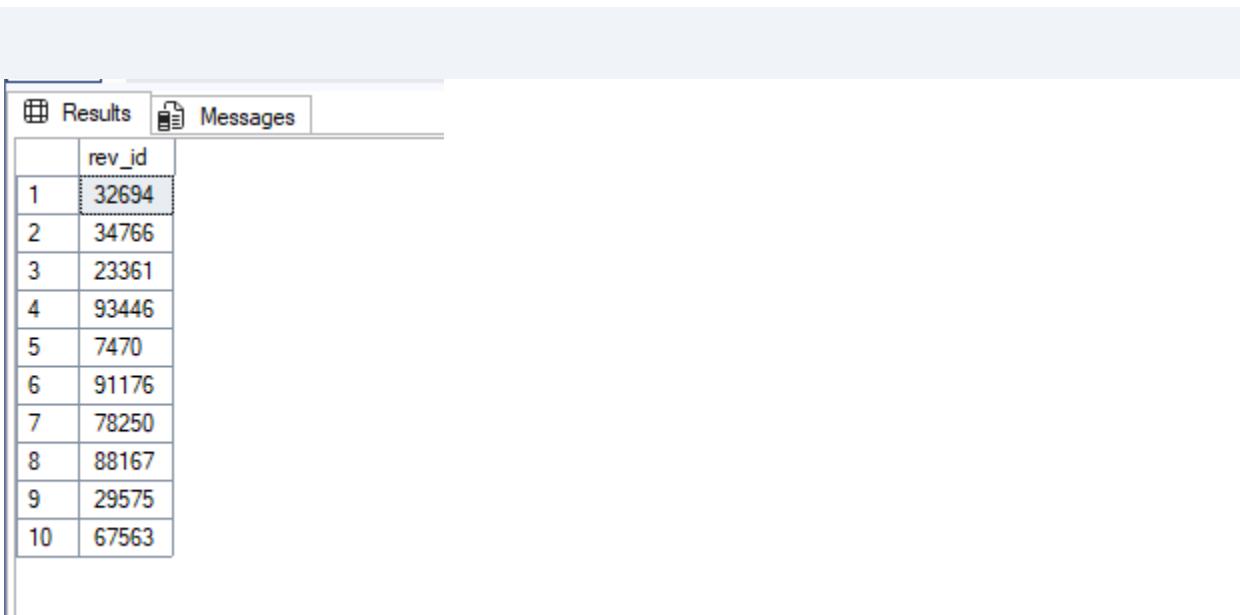
DECLARE @offset INT = @l - 1; -- How many to skip

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running IQ5: Paginated k-NN with Date Filter (Slow). ---';

-- 3. Run the query
SELECT
    r.rev_id
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
    -- Pre-filter (Must happen before sorting)
    r.rev_timestamp >= @date_low
    AND r.rev_timestamp <= @date_high
ORDER BY
    -- Full scan required to sort filtered results
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC,
    t.old_id ASC
-- Standard SQL Pagination
OFFSET @offset ROWS
FETCH NEXT @limit ROWS ONLY;
GO

```



The screenshot shows the SSMS Results pane with two tabs: 'Results' (selected) and 'Messages'. The 'Results' tab displays a table with one column, 'rev\_id', containing 10 rows of data.

	rev_id
1	32694
2	34766
3	23361
4	93446
5	7470
6	91176
7	78250
8	88167
9	29575
10	67563

## Q21 IQ6

```
SELECT rev_id
FROM text JOIN revision ON old_id=rev_id
WHERE text_embedding {op} '{q}' BETWEEN {d} AND {d*} AND
rev_timestamp>='{{DATE_LOW}}' AND rev_timestamp<='{{DATE_HIGH}}'
ORDER BY text_embedding {op} '{q}', old_id;
```

### Explanation of IQ6

This is a **Filtered Bounded Range Search with Join**.

1. **Join:** It joins the `text` and `revision` tables.
2. **Relational Filter:** It restricts the search to revisions within a specific date range (`rev_timestamp`).
3. **Vector Range Filter:** It selects text segments where the semantic distance to the query vector falls within a specific band (between `{d}` and `{d*}`).
4. **Sorting:** It orders the results by similarity distance.

### Indexed Translation: Not Possible

Just like with **IQ2** and **IQ4**, this is a **Range Search** (specifically a bounded/annulus search).

- The `VECTOR_SEARCH` index function strictly requires a `TOP_N` parameter. It cannot return "all items within a specific distance band."
- Therefore, we **cannot** use the vector index. We must use the scalar `VECTOR_DISTANCE` function in the `WHERE` clause, forcing a full table scan.

### IQ6 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

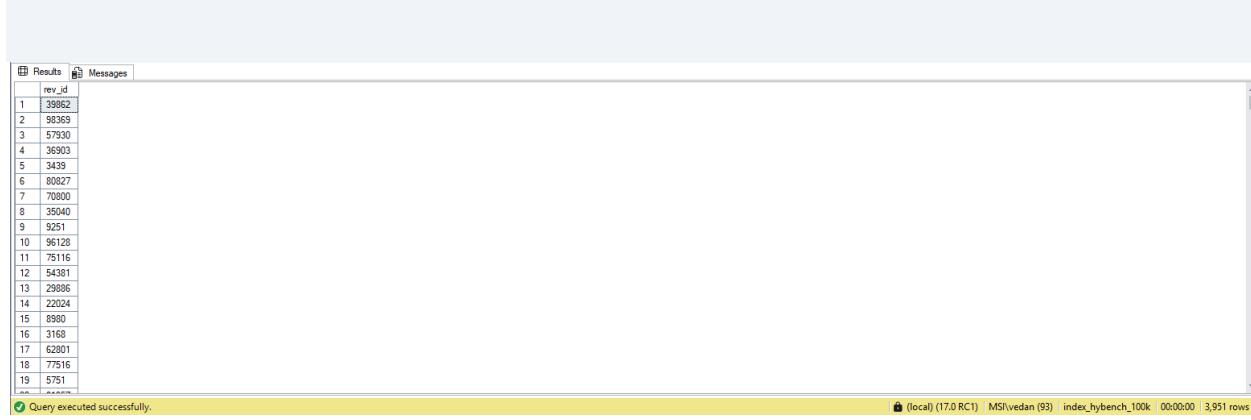
-- 1. Define parameters
DECLARE @dist_min FLOAT = 0.2;      -- {d}
DECLARE @dist_max FLOAT = 0.5;      -- {d*}
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z'; -- {DATE_LOW}
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z'; -- {DATE_HIGH}
DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running IQ6: Filtered Bounded Range Search (Slow). ---';
```

```
-- 3. Run the query
SELECT
    r.rev_id
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
    -- Relational Filter: Date Range
    r.rev_timestamp >= @date_low
    AND r.rev_timestamp <= @date_high
    -- Vector Range Filter (Index cannot be used)
    AND VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @dist_min
AND @dist_max
ORDER BY
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC,
    t.old_id ASC;
```

GO



The screenshot shows a SQL query results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is selected and displays a table with one column, 'rev\_id', containing 19 integer values. The 'Messages' tab is visible but empty.

rev_id
1 39852
2 98359
3 57930
4 36903
5 3439
6 80827
7 70800
8 35040
9 9251
10 96128
11 75116
12 54381
13 29886
14 22024
15 8980
16 3168
17 62801
18 77516
19 5751

Query executed successfully.

(local) (17.0 RC1) | MSIvedan (93) | index\_hybench\_100k | 00:00:00 | 3,951 rows

## Q22 IQ7:

```
SELECT rev_actor,COUNT(*) AS cou
FROM (
    SELECT page_id
    FROM page
    WHERE page_len < {len}
    ORDER BY page_embedding {op} '{q}', page_id
    LIMIT {r-l+1} OFFSET {l-1}
) AS new_page JOIN revision
ON page_id=rev_page
GROUP BY rev_actor
```

```
ORDER BY cou DESC;
```

IQ7

## Explanation of IQ7

This is an **Aggregation on a Paginated k-NN Search with a Pre-filter**.

1. **Inner Query (Search):**
  - o **Filter:** Selects only pages shorter than {len}.
  - o **Rank:** Orders them by semantic similarity to the query vector.
  - o **Paginate:** Retrieves a specific slice of results (ranks {l} to {r}).
2. **Outer Query (Aggregation):**
  - o Joins those specific pages to the revision table.
  - o Counts the contributions per author (rev\_actor).

## Indexed Translation: Not Possible

Just like **IQ3**, this query combines a **Pre-filter** (page\_len < {len}) with **Pagination**.

- **The Issue:** The `VECTOR_SEARCH` index function is a **Post-filter**. It finds the global top matches first. If the global top matches are all "long" pages, `VECTOR_SEARCH` would return them, the `WHERE` clause would discard them, and you would end up with an empty or incomplete page of results.
- **The Fix:** To guarantee you get a full page of "short" articles, we must filter first and then sort. This requires using `VECTOR_DISTANCE` in the `ORDER BY` clause, forcing a full table scan.

## IQ7 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB  
GO
```

```
-- 1. Define parameters
```

```
DECLARE @page_len_limit INT = 1000; -- {len}  
DECLARE @l INT = 11; -- Start Rank {l}  
DECLARE @r INT = 20; -- End Rank {r}  
DECLARE @query_vector VECTOR(384);
```

```
-- Calculate SQL pagination arguments
```

```
DECLARE @limit INT = @r - @l + 1; -- {r-l+1} (How many to fetch)  
DECLARE @offset INT = @l - 1; -- {l-1} (How many to skip)
```

```
-- 2. Grab a real vector
```

```

SELECT TOP 1 @query_vector = page_embedding FROM dbo.page ORDER BY page_id;

PRINT '--- Running IQ7: Aggregation on Paginated Pre-filtered Search (Slow). ---';

-- 3. Run the query
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM
(
    -- Inner Query: Paginated k-NN with Pre-filter
    SELECT
        p.page_id
    FROM
        dbo.page AS p
    WHERE
        p.page_len < @page_len_limit -- Pre-filter
    ORDER BY
        VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
        p.page_id ASC
    OFFSET @offset ROWS
    FETCH NEXT @limit ROWS ONLY
) AS new_page
JOIN
    dbo.revision AS r ON new_page.page_id = r.rev_page
GROUP BY
    r.rev_user_text
ORDER BY
    cou DESC;
GO

```

100 % No issues found

	rev_actor	cou
1	7	1
2	4096	1
3	51	1
4	1441	1
5	10807	1
6	200	1
7	2121	1
8	5	1
9	7560	1
10	95	1

Q23 IQ8:

```
SELECT rev_actor,COUNT(*) AS cou
FROM page JOIN revision ON page_id=rev_page
WHERE page_len < {len} AND page_embedding {op} '{q}' BETWEEN {d} AND {d*}
GROUP BY rev_actor
ORDER BY cou DESC;
```

### Explanation of IQ8

This is an **Aggregation on a Filtered Bounded Range Search**.

1. **Filtering:** It selects pages that meet two criteria:
  - o **Relational:** Length is less than {len}.
  - o **Vector:** Semantic distance to the query vector is within a specific band (between {d} and {d\*}).
2. **Joining & Aggregating:** It joins matching pages to the revision table and counts the number of revisions per author (rev\_actor).

### Indexed Translation: Not Possible

This is a **Bounded Range Search** (finding all items within a distance range).

- The VECTOR\_SEARCH index function strictly requires a TOP\_N limit. It cannot return "all items between distance X and Y."
- Therefore, we **cannot** use the vector index. We must use VECTOR\_DISTANCE in the WHERE clause, which forces a full table scan.

### IQ8 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

```
-- 1. Define parameters
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @dist_min FLOAT = 0.2; -- {d}
DECLARE @dist_max FLOAT = 0.5; -- {d*}
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running IQ8: Aggregation on Filtered Bounded Range Search (Slow). ---';
```

```
-- 3. Run the query
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM
    dbo.page AS p
JOIN
    dbo.revision AS r ON p.page_id = r.rev_page
WHERE
    -- Relational Filter
    p.page_len < @page_len_limit
    -- Vector Range Filter (Index cannot be used)
    AND VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @dist_min
    AND @dist_max
GROUP BY
    r.rev_user_text
ORDER BY
    cou DESC;
GO
```

The screenshot shows a SQL query results window with a table titled 'Results'. The table has two columns: 'rev\_actor' and 'cou'. The data is as follows:

rev_actor	cou
1	8
2	5
3	7
4	21
5	38
6	34
7	650
8	24
9	53
10	31
11	30
12	20
13	532
14	51
15	123
16	233
17	9
18	746
19	25
20	98
21	44
22	109
23	106
24	103
25	100
26	99
27	100
28	100
29	100
30	100
31	100
32	100
33	100
34	100
35	100
36	100
37	100
38	100
39	100
40	100
41	100
42	100
43	100
44	100
45	100
46	100
47	100
48	100
49	100
50	100
51	100
52	100
53	100
54	100
55	100
56	100
57	100
58	100
59	100
60	100
61	100
62	100
63	100
64	100
65	100
66	100
67	100
68	100
69	100
70	100
71	100
72	100
73	100
74	100
75	100
76	100
77	100
78	100
79	100
80	100
81	100
82	100
83	100
84	100
85	100
86	100
87	100
88	100
89	100
90	100
91	100
92	100
93	100
94	100
95	100
96	100
97	100
98	100
99	100
100	100
101	100
102	100
103	100
104	100
105	100
106	100
107	100
108	100
109	100
110	100
111	100
112	100
113	100
114	100
115	100
116	100
117	100
118	100
119	100
120	100
121	100
122	100
123	100
124	100
125	100
126	100
127	100
128	100
129	100
130	100
131	100
132	100
133	100
134	100
135	100
136	100
137	100
138	100
139	100
140	100
141	100
142	100
143	100
144	100
145	100
146	100
147	100
148	100
149	100
150	100
151	100
152	100
153	100
154	100
155	100
156	100
157	100
158	100
159	100
160	100
161	100
162	100
163	100
164	100
165	100
166	100
167	100
168	100
169	100
170	100
171	100
172	100
173	100
174	100
175	100
176	100
177	100
178	100
179	100
180	100
181	100
182	100
183	100
184	100
185	100
186	100
187	100
188	100
189	100
190	100
191	100
192	100
193	100
194	100
195	100
196	100
197	100
198	100
199	100
200	100
201	100
202	100
203	100
204	100
205	100
206	100
207	100
208	100
209	100
210	100
211	100
212	100
213	100
214	100
215	100
216	100
217	100
218	100
219	100
220	100
221	100
222	100
223	100
224	100
225	100
226	100
227	100
228	100
229	100
230	100
231	100
232	100
233	100
234	100
235	100
236	100
237	100
238	100
239	100
240	100
241	100
242	100
243	100
244	100
245	100
246	100
247	100
248	100
249	100
250	100
251	100
252	100
253	100
254	100
255	100
256	100
257	100
258	100
259	100
260	100
261	100
262	100
263	100
264	100
265	100
266	100
267	100
268	100
269	100
270	100
271	100
272	100
273	100
274	100
275	100
276	100
277	100
278	100
279	100
280	100
281	100
282	100
283	100
284	100
285	100
286	100
287	100
288	100
289	100
290	100
291	100
292	100
293	100
294	100
295	100
296	100
297	100
298	100
299	100
300	100
301	100
302	100
303	100
304	100
305	100
306	100
307	100
308	100
309	100
310	100
311	100
312	100
313	100
314	100
315	100
316	100
317	100
318	100
319	100
320	100
321	100
322	100
323	100
324	100
325	100
326	100
327	100
328	100
329	100
330	100
331	100
332	100
333	100
334	100
335	100
336	100
337	100
338	100
339	100
340	100
341	100
342	100
343	100
344	100
345	100
346	100
347	100
348	100
349	100
350	100
351	100
352	100
353	100
354	100
355	100
356	100
357	100
358	100
359	100
360	100
361	100
362	100
363	100
364	100
365	100
366	100
367	100
368	100
369	100
370	100
371	100
372	100
373	100
374	100
375	100
376	100
377	100
378	100
379	100
380	100
381	100
382	100
383	100
384	100
385	100
386	100
387	100
388	100
389	100
390	100
391	100
392	100
393	100
394	100
395	100
396	100
397	100
398	100
399	100
400	100
401	100
402	100
403	100
404	100
405	100
406	100
407	100
408	100
409	100
410	100
411	100
412	100
413	100
414	100
415	100
416	100
417	100
418	100
419	100
420	100
421	100
422	100
423	100
424	100
425	100
426	100
427	100
428	100
429	100
430	100
431	100
432	100
433	100
434	100
435	100
436	100
437	100
438	100
439	100
440	100
441	100
442	100
443	100
444	100
445	100
446	100
447	100
448	100
449	100
450	100
451	100
452	100
453	100
454	100
455	100
456	100
457	100
458	100
459	100
460	100
461	100
462	100
463	100
464	100
465	100
466	100
467	100
468	100
469	100
470	100
471	100
472	100
473	100
474	100
475	100
476	100
477	100
478	100
479	100
480	100
481	100
482	100
483	100
484	100
485	100
486	100
487	100
488	100
489	100
490	100
491	100
492	100
493	100
494	100
495	100
496	100
497	100
498	100
499	100
500	100
501	100
502	100
503	100
504	100
505	100
506	100
507	100
508	100
509	100
510	100
511	100
512	100
513	100
514	100
515	100
516	100
517	100
518	100
519	100
520	100
521	100
522	100
523	100
524	100
525	100
526	100
527	100
528	100
529	100
530	100
531	100
532	100
533	100
534	100
535	100
536	100
537	100
538	100
539	100
540	100
541	100
542	100
543	100
544	100
545	100
546	100
547	100
548	100
549	100
550	100
551	100
552	100
553	100
554	100
555	100
556	100
557	100
558	100
559	100
560	100
561	100
562	10

```
ORDER BY year DESC,  
distance ASC;
```

## Explanation of IQ9

This is a **Paginated Partitioned k-NN Search**.

1. **Filtering:** It selects revisions within a specific year range (`{YEARL}` to `{YEARH}`).
2. **Partitioning:** It groups the results by `year`.
3. **Ranking:** Inside *each* year group, it ranks the text segments by their similarity to the query vector.
4. **Pagination:** Unlike NQ9 (which took the "Top K"), this query takes a **specific slice** of ranks (e.g., "ranks 11 to 20") for *each* year.

## Indexed Translation: Not Possible

The `VECTOR_SEARCH` index function finds the **global** top N matches. It cannot handle:

1. **Partitioning:** Finding the top `N per category` (year).
2. **Offset/Slicing:** Finding *only* ranks `{l}` through `{r}` without retrieving the top ranks first.

To achieve "per-year" ranking, we must use a window function (`ROW_NUMBER()`) combined with `VECTOR_DISTANCE`, which requires calculating the distance for all matching rows (a table scan).

## IQ9 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB  
GO  
  
-- 1. Define parameters  
DECLARE @l INT = 11; -- Start Rank {l}  
DECLARE @r INT = 20; -- End Rank {r}  
DECLARE @year_low INT = 2010; -- {YEARL}  
DECLARE @year_high INT = 2015; -- {YEARH}  
DECLARE @query_vector VECTOR(384);  
  
-- 2. Grab a real vector  
SELECT TOP 1 @query_vector = text_embedding  
FROM dbo.text  
WHERE text_embedding IS NOT NULL  
ORDER BY old_id;  
  
PRINT '--- Running IQ9: Paginated Partitioned k-NN (Slow). ---';
```

```

-- 3. Run the query
SELECT
    ranked_pages.[year],
    ranked_pages.old_id,
    ranked_pages.distance
FROM (
    SELECT
        t.old_id,
        LEFT(r.rev_timestamp, 4) AS [year], -- Robust Year Extraction
        VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance,
        -- Partition by Year and Rank by Distance
        ROW_NUMBER() OVER (
            PARTITION BY LEFT(r.rev_timestamp, 4)
            ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC, t.old_id
        ) AS rank
    FROM
        dbo.text AS t
    JOIN
        dbo.revision AS r ON t.old_id = r.rev_text_id
    WHERE
        -- Filter date range first to reduce rows for the window function
        CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
) AS ranked_pages
WHERE
    ranked_pages.rank BETWEEN @l AND @r -- Select the specific slice
ORDER BY
    ranked_pages.[year] DESC,
    ranked_pages.distance ASC;
GO

```

The screenshot shows a SQL query results window with the following details:

- Results Tab:** The active tab, showing the query results.
- Messages Tab:** An empty tab.
- Header:** Includes a zoom icon, a green circular icon with a checkmark, and the text "No issues found".
- Toolbar:** Includes icons for copy, cut, paste, and refresh, followed by "Ln: 40 Ch: 72 SPC CRLF".
- Table Data:**

	year	old_id	distance
1	2015	37528	0.054505801353455
2	2015	53774	0.055388212039795
3	2015	26712	0.055412113665344
4	2015	98412	0.05542640505023193
5	2015	9917	0.055515607719421
6	2015	3638	0.0555764504432678
7	2015	60698	0.0564698769020081
8	2015	40953	0.0571471452713013
9	2015	32822	0.0572123527526855
10	2015	907	0.0572460293769836
11	2014	43174	0.061635434627533
12	2014	7447	0.0620973706245422
13	2014	57129	0.0634285807609558
14	2014	2926	0.0635675191879272
15	2014	499	0.0636052469280701
16	2014	75341	0.0642106533050537
17	2014	61591	0.0644053155860901
18	2014	73181	0.0644727349281311
19	2014	69465	0.0645948052406311
- Status Bar:** Shows a green checkmark icon, the text "Query executed successfully.", and the status "(local) (17.0 RC1) MSI\vedan (93) index\_hybench\_100k 00:00:00 60 rows".

## Q25 IQ 10:

```
SELECT year, old_id, distance
FROM (
    SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding
    {op} '{q}' AS distance
    FROM text JOIN revision ON old_id = rev_id
    WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND
    {YEARH} AND text_embedding {op} '{q}' BETWEEN {d} AND {d*}
) AS ranked_pages
ORDER BY year DESC,
distance ASC;
```

### Explanation of IQ10

This is a **Filtered Bounded Range Search**.

1. **Filtering (Relational)**: It selects text revisions within a specific year range ({YEARL} to {YEARH}).
2. **Filtering (Vector)**: It selects text segments where the semantic distance to the query vector falls within a specific band (between {d} and {d\*}).
3. **Sorting**: It orders the results by year and then by similarity distance.

### Indexed Translation: Not Possible

This is a **Bounded Range Search** (finding all items within a distance band).

- The `VECTOR_SEARCH` index function strictly requires a `TOP_N` limit. It cannot return "all items between distance X and Y."
- Therefore, we **cannot** use the vector index. We must use `VECTOR_DISTANCE` in the `WHERE` clause, which forces a full table scan.

### IQ10 Translation (Non-Indexed)

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

```
-- 1. Define parameters
DECLARE @dist_min FLOAT = 0.2;      -- {d}
DECLARE @dist_max FLOAT = 0.5;      -- {d*}
DECLARE @year_low INT = 2010;        -- {YEARL}
DECLARE @year_high INT = 2015;       -- {YEARH}
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```

SELECT TOP 1 @query_vector = text_embedding
FROM dbo.text
WHERE text_embedding IS NOT NULL
ORDER BY old_id;

```

PRINT '--- Running IQ10: Filtered Bounded Range Search (Slow). ---';

-- 3. Run the query

```

SELECT
    LEFT(r.rev_timestamp, 4) AS [year], -- Robust Year Extraction
    t.old_id,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
    -- Filter 1: Date Range
    CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
    -- Filter 2: Vector Range (Index cannot be used)
    AND VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @dist_min
    AND @dist_max
ORDER BY
    [year] DESC,
    distance ASC;
GO

```

	year	old_id	distance
1	2015	3410	0.20098872184753
2	2015	45567	0.20048838214974
3	2015	91	0.20217323302227
4	2015	87531	0.202220261056954
5	2015	60824	0.205597823181152
6	2015	35393	0.206443786621094
7	2015	36175	0.207675099372864
8	2015	398	0.207916915416718
9	2015	40379	0.208791017523249
10	2015	40350	0.208791017523249
11	2015	40384	0.208791017523249
12	2015	40355	0.208791017523249
13	2015	40362	0.208791017523249
14	2015	40369	0.208791017523249
15	2015	36116	0.20932012796402
16	2015	12324	0.2105010139038
17	2015	50536	0.211006879806519
18	2015	31845	0.21159583301544
19	2015	12295	0.212383151054382

Query executed successfully.

(local) (17.0 RC1) | MSI\vedan (81) | index\_hybench\_100k | 00:00:01 | 4,506 rows

# SQ - SQLServer

## Q26 SQ1

```
SELECT *
FROM (
    SELECT old_id, old_text, ROW_NUMBER() OVER () AS rank
    FROM (
        SELECT old_id, old_text
        FROM text
        ORDER BY text_embedding {op} '{q}'
    ) AS ordered_limited
) AS ranked
WHERE rank IN ({r_1}, {r_2}, ..., {r_n});
```

### Explanation of SQ1

This is a **Specific Rank Selection Query**.

- **Logic:** It ranks *all* text segments by semantic similarity to the query vector and then retrieves only the items at specific positions (ranks), such as "the 1st, the 5th, and the 10th match."
- **Use Case:** This is often used for probing the quality of search results at different depths without fetching the entire list.

### Indexed Translation: Not Possible

We **cannot** effectively use the vector index (VECTOR\_SEARCH) here.

- **Reason:** The index is designed to return the **Top N** contiguous items (1, 2, 3... N). It cannot natively "skip" to pick specific ranks like 50 or 100 without fetching everything before them.
- **Current Status:** Since your index creation is blocked by the bug anyway, we must use the VECTOR\_DISTANCE function with a standard ROW\_NUMBER() window function. This forces a full table scan and sort.

### SQ1 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

```
-- 1. Define parameters
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector to search with
```

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```
PRINT '--- Running SQ1: Specific Rank Selection (Slow). ---';
```

### -- 3. Run the query

## SELECT

ranked.old\_id,  
ranked.old\_text,  
ranked.rank

FROM (

## SELECT

t.old id,

t.old text,

-- Calculate Rank based on Distance

ROW\_NUMBER() OVER (

```
ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC, t.old_id
```

ASC

) AS rank

FROM

dbo.text AS t

) AS ranked

WHERE

-- Filte

-- Replace this list with your specific {r} values  
rank IN (1, 5, 10, 50, 100)

ORDER BY

rank ASC

Q

Result

	old_id	old_text	rank
1	1	<strong>MediaWiki has been installed.</strong>\n\n	1
2	28262	==Wikimedia Pennyvania==\nHello there!\n\n	5
3	94952	((WikiProject banner shell class=Stub))\n((WikiProj...	10
4	10087	Please do not remove content from Wikipedia. It is c...	50
5	13445	((WikiProject banner shell class=Stub))\n((WikiProj...	100

Q27 SQ2

## Explanation of SQ2

This is a **Multi-Range Search** (or Multi-Annulus Search).

- **Logic:** Instead of looking for items within a single distance band (like IQ2), this query looks for items that fall into *any* of several specific distance bands (e.g., "very close" OR "moderately far").
- **Use Case:** This might be used to sample data from different strata of similarity (e.g., finding examples of "exact matches," "near matches," and "far matches" in one go).

## Indexed Translation: Not Possible

As with all other **Range Searches** we have encountered (Q2, Q6, IQ2, etc.), we **cannot** use the vector index (VECTOR\_SEARCH) here.

- **Reason:** The index requires a strict TOP\_N limit. It does not support "distance thresholds," let alone *multiple* disjoint thresholds.
- **Impact:** We must use VECTOR\_DISTANCE in the WHERE clause with multiple OR conditions. This forces a full table scan.

## SQ2 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
-- Range 1
DECLARE @d1_min FLOAT = 0.1;
DECLARE @d1_max FLOAT = 0.2;
-- Range 2
DECLARE @d2_min FLOAT = 0.4;
DECLARE @d2_max FLOAT = 0.5;
-- Query Vector
DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running SQ2: Multi-Range Search (Slow). ---';

-- 3. Run the query
SELECT
    t.old_id,
    t.old_text,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
WHERE
```

```
-- Range 1
(VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d1_min AND
@d1_max)
OR
-- Range 2
(VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d2_min AND
@d2_max)
-- Add more OR conditions as needed
ORDER BY
    distance ASC,
    t.old_id ASC;
GO
```

## Q28 SQ3

```
SELECT *
FROM (
    SELECT page_id, page_title, ROW_NUMBER() OVER () AS rank
    FROM (
        SELECT page_id, page_title
        FROM page
        WHERE page_len < {len}
        ORDER BY page_embedding {op} '{q}', page_id
    ) AS top_k
) AS ranked
WHERE rank IN ({r_1}, {r_2}, .., {r_n});
```

## Explanation of SQ3

This is a **Specific Rank Selection with Pre-filter**.

- Logic:

1. **Filter:** Selects only pages shorter than `{len}`.
  2. **Rank:** Orders the remaining pages by semantic similarity to the query vector.
  3. **Select:** Retrieves only specific positions from this ranked list (e.g., "Get the 1st, 10th, and 50th matching short page").
- **Use Case:** Checking result quality at specific depths for a filtered subset of data.

## Indexed Translation: Not Possible

Just like **Q3** and **IQ3**, this query has a **Pre-filter** (`page_len < {len}`).

- **The Issue:** The `VECTOR_SEARCH` index function is a **Post-filter**. It finds the global top matches first. If the global top matches are all "long" pages, they get filtered out later, destabilizing the ranking. You can't ask the index for "The 10th best *short* page."
- **The Fix:** We must scan the table, filter by length, calculate distances for the survivors, and then rank them.

## SQ3 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector to search with

```
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running SQ3: Specific Rank Selection with Pre-Filter (Slow). ---';
```

-- 3. Run the query

```
SELECT
    ranked.page_id,
    ranked.page_title,
    ranked.rank
FROM (
    SELECT
        p.page_id,
        p.page_title,
        -- Calculate Rank based on Distance
        ROW_NUMBER() OVER (
```

```

        ORDER BY VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
p.page_id ASC
    ) AS rank
FROM
    dbo.page AS p
WHERE
    -- Pre-filter: Only consider short pages
    p.page_len < @page_len_limit
) AS ranked
WHERE
    -- Filter for specific ranks {r_1, r_2, ...}
    rank IN (1, 5, 10, 50, 100)
ORDER BY
    rank ASC;
GO

```

page_id	page_title	rank
1	Main_Page	1
2	Start_bt	5
3	Screen_angle	10
4	Scratch_board	50
5	Google_news	100

## Q29 SQ4

```

SELECT page_id, page_title,
FROM page
WHERE page_len < {len} AND ((page_embedding {op} '{q}' BETWEEN {d_1} AND {d_1*})
OR (page_embedding {op} '{q}' BETWEEN {d_2} AND {d_2*})
OR .. OR (page_embedding {op} '{q}' BETWEEN {d_n} AND {d_n*}))
ORDER BY page_embedding {op} '{q}', page_id;

```

### Explanation of SQ4

This is a **Filtered Multi-Range Search**.

- **Logic:**
  1. **Relational Filter:** Selects only pages shorter than `{len}`.
  2. **Vector Multi-Range Filter:** Selects pages where the semantic distance falls into *any* of several specific bands (e.g., "very close" OR "somewhat far").
  3. **Sorting:** Orders results by distance.
- **Use Case:** Complex sampling or categorizing items into specific similarity buckets while excluding long documents.

## Indexed Translation: Not Possible

We cannot use the vector index (VECTOR\_SEARCH) for this query.

- **Reason 1 (Range Search):** The index requires a TOP\_N limit. It cannot handle "distance between X and Y".
- **Reason 2 (Pre-filter):** The index cannot apply the page\_len < {len} filter before searching the graph.
- **Reason 3 (Complex Logic):** The index cannot natively handle the OR logic between multiple distance ranges.

We must use VECTOR\_DISTANCE in the WHERE clause, forcing a full table scan.

## SQ4 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @query_vector VECTOR(384);
```

-- Range 1 ({d\_1} to {d\_1\*})

```
DECLARE @d1_min FLOAT = 0.1;
DECLARE @d1_max FLOAT = 0.2;
```

-- Range 2 ({d\_2} to {d\_2\*})

```
DECLARE @d2_min FLOAT = 0.4;
DECLARE @d2_max FLOAT = 0.5;
```

-- 2. Grab a real vector to search with

```
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running SQ4: Filtered Multi-Range Search (Slow). ---';
```

-- 3. Run the query

```
SELECT
    p.page_id,
    p.page_title,
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) AS distance
FROM
    dbo.page AS p
```

## WHERE

-- Filter 1: Relational (Pre-filter)  
p.page\_len < @page\_len\_limit

## AND (

-- Filter 2: Vector Range 1

(VECTOR\_DISTANCE('cosine', @query\_vector, p.page\_embedding) BETWEEN @d1\_min  
AND @d1\_max)

## OR

-- Filter 3: Vector Range 2

(VECTOR\_DISTANCE('cosine', @query\_vector, p.page\_embedding) BETWEEN @d2\_min  
AND @d2\_max)

)

## ORDER BY

distance ASC,  
p.page\_id ASC;

GO

	Results	Messages
1	page_id page_title distance	
1	62797 Volt42	0.100001215934753
2	36339 MVM_Security	0.10000866651535
3	19177 Xavier_Davis	0.100014626979828
4	85995 Anbar_province	0.100016713142395
5	33352 Mathew_George	0.100017607212067
6	76836 Los_Conner	0.100018501281738
7	4618 Wobblies_world	0.100042619976807
8	81099 Paul_Sundeland	0.100055932998657
9	9101 Latent_nasal_branch	0.100057244300842
10	27384 Angel_Hernandez	0.10006183385849
11	27393 Angel_Hernandez	0.10006183385849
12	58355 Wave_pro	0.100064098834991
13	48322 Killa_Deer	0.100077211056642
14	37612 Canadian_Back_Watch	0.10008579452569
15	78129 Gay_Thompson	0.100092649459839
16	49333 Cloaked_Dagger	0.100092847483063
17	00940 Biblical_Astronomy	0.100095464157104
18	78181 Adam_Malk	0.10009593598633
19	78302 Ian_Armstrong	0.100119629177656

Query executed successfully.

(local) (17.0 RC1) | MSI\vedan (B1) | index\_hybench\_100k | 00:00:00 | 32,838 rows

## Q30 SQ5

```
SELECT rev_id
FROM (
    SELECT rev_id, ROW_NUMBER() OVER () AS rank
    FROM (
        SELECT rev_id
        FROM text JOIN revision ON old_id = rev_id
        WHERE rev_timestamp >= '{DATE_LOW}' AND rev_timestamp <= '{DATE_HIGH}'
        ORDER BY text_embedding {op} '{q}', old_id
    ) AS top_k
) AS ranked
WHERE rank IN ({r_1}, {r_2}, ..., {r_n});
```

## Explanation of SQ5

This is a **Specific Rank Selection on a Filtered Join**.

- Filtering (Relational):** It selects revisions within a specific date range (`{DATE_LOW}` to `{DATE_HIGH}`).
- Ranking:** It ranks the matching revisions by the semantic similarity of their text content to the query vector.
- Select Specific Ranks:** It returns only specific positions from this ranked list (e.g., "the 1st, 10th, and 100th matching revision in that date range").

## Indexed Translation: Not Possible

We **cannot** use the vector index (`VECTOR_SEARCH`) for this query.

- Reason 1 (Pre-filter):** The index is a Post-filter. It finds the global top matches first. If the global top matches fall outside the date range, `VECTOR_SEARCH` discards them, breaking the ranking logic for the specific date range.
- Reason 2 (Rank Selection):** The index cannot "jump" to specific ranks (e.g., "Get rank 50") without retrieving ranks 1-49 first.

We must use `VECTOR_DISTANCE` in the `ORDER BY` clause inside a window function, which forces a scan of all rows matching the date filter.

## SQ5 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z'; -- {DATE_LOW}
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z'; -- {DATE_HIGH}
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```
PRINT '--- Running SQ5: Specific Rank Selection with Date Filter (Slow). ---';
```

-- 3. Run the query

```
SELECT
    ranked.rev_id,
    ranked.rank
FROM (
    SELECT
        r.rev_id,
        -- Calculate Rank based on Distance
        ROW_NUMBER() OVER (
```

```

        ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC, t.old_id
ASC
    ) AS rank
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
    -- Pre-filter: Date Range
    r.rev_timestamp >= @date_low
    AND r.rev_timestamp <= @date_high
) AS ranked
WHERE
    -- Filter for specific ranks {r_1, r_2, ...}
    rank IN (1, 5, 10, 50, 100)
ORDER BY
    rank ASC;
GO

```

	rev_id	rank
1	18443	1
2	49408	5
3	22769	10
4	96759	50
5	96767	100

Q31 SQ6:

```

SELECT rev_id
FROM text JOIN revision ON old_id = rev_id
WHERE rev_timestamp >= '{DATE_LOW}' AND rev_timestamp <= '{DATE_HIGH}'
AND ((text_embedding {op} '{q}' BETWEEN {d_1} AND {d_1*})
OR (text_embedding {op} '{q}' BETWEEN {d_2} AND {d_2*})
OR .. OR (text_embedding {op} '{q}' BETWEEN {d_n} AND {d_n*}))
ORDER BY text_embedding {op} '{q}', old_id;
USE index_hybench_100k; -- Or HyBenchDB
GO

```

-- 1. Define parameters

```

DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z'; -- {DATE_LOW}
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z'; -- {DATE_HIGH}
DECLARE @query_vector VECTOR(384);

-- Range 1 ({d_1} to {d_1*})
DECLARE @d1_min FLOAT = 0.1;
DECLARE @d1_max FLOAT = 0.2;

-- Range 2 ({d_2} to {d_2*})
DECLARE @d2_min FLOAT = 0.4;
DECLARE @d2_max FLOAT = 0.5;

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running SQ6: Filtered Multi-Range Search with Join (Slow). ---';

-- 3. Run the query
SELECT
    r.rev_id
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
    -- Filter 1: Relational Date Range
    r.rev_timestamp >= @date_low
    AND r.rev_timestamp <= @date_high

    AND (
        -- Filter 2: Vector Range 1
        (VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d1_min
        AND @d1_max)
        OR
        -- Filter 3: Vector Range 2
        (VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d2_min
        AND @d2_max)
    )
ORDER BY
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC,
    t.old_id ASC;
GO

```

rev_id
1 25450
2 30518
3 20713
4 87405
5 56101
6 89804
7 43459
8 69292
9 48535
10 7559
11 3405
12 29195
13 19467
14 49066
15 96973
16 63567
17 2176
18 83178
19 12831

Q32 SQ7:

```

SELECT rev_actor, COUNT(*) AS cou
FROM (
    SELECT page_id
    FROM (
        SELECT page_id, ROW_NUMBER() OVER () AS rank
        FROM (
            SELECT page_id
            FROM page
            WHERE page_len < {len}
            ORDER BY page_embedding {op} '{q}', page_id
        ) AS top_k
    ) AS ranked
    WHERE rank IN ({r_1}, {r_2}, ..., {r_n})
) AS filtered_page JOIN revision ON page_id = rev_page
GROUP BY rev_actor
ORDER BY cou DESC;

```

## Explanation of SQ7

This is an **Aggregation on Specific Rank Selection with a Pre-filter**.

- Filter & Rank (Inner):**
  - Pre-filter:** It selects only pages shorter than `{len}`.
  - Rank:** It orders these pages by semantic similarity to the query vector.
  - Select Ranks:** It picks specific positions from this filtered list (e.g., "the 1st, 10th, and 50th best matching *short* page").
- Join & Aggregate (Outer):** It joins those specific pages to the `revision` table to count the contributions of each author (`rev_actor`).

## Indexed Translation: Not Possible

We **cannot** use the vector index (`VECTOR_SEARCH`) for this query.

- **Reason:** The index is a **Post-filter**. It finds the global top N matches first. If you ask for "Rank 1," the index finds the single closest page in the database. If that page happens to be *long* ( $> \{\text{len}\}$ ), it gets filtered out, and you are left with **Rank 1: NULL**. The index cannot "skip" the long pages during its search to find the "Rank 1 Short Page."
- **The Fix:** We must scan the table, apply the length filter first, and then sort the remaining rows to calculate the correct ranks.

## SQ7 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding FROM dbo.page ORDER BY page_id;

PRINT '--- Running SQ7: Aggregation on Specific Ranks with Pre-Filter (Slow). ---';

-- 3. Run the query
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM
(
    SELECT
        ranked.page_id
    FROM (
        SELECT
            p.page_id,
            -- Calculate Rank based on Distance, considering only short pages
            ROW_NUMBER() OVER (
                ORDER BY VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
                p.page_id ASC
            ) AS rank
        FROM
            dbo.page AS p
        WHERE
            p.page_len < @page_len_limit -- Pre-filter
    ) AS ranked
    WHERE
        -- Select specific ranks {r_1, r_2...}
```

```

        ranked.rank IN (1, 5, 10, 50, 100)
    ) AS filtered_page
JOIN
    dbo.revision AS r ON filtered_page.page_id = r.rev_page
GROUP BY
    r.rev_user_text
ORDER BY
    cou DESC;
GO

```

	rev_actor	cou	
1	10343	1	1
2	7531	1	1
3	30	1	1
4	8755	1	1
5	2	1	1

Q33 SQ8:

```

SELECT rev_actor, COUNT(*) AS cou
FROM (
    SELECT page_id
    FROM page
    WHERE page_len < {len} AND
        ((page_embedding {op} '{q}' BETWEEN {d_1} AND {d_1*}) OR (page_embedding {op} '{q}' BETWEEN {d_2} AND {d_2*}) OR .. OR (page_embedding {op} '{q}' BETWEEN {d_n} AND {d_n*}))
) AS filtered_page
JOIN revision ON page_id = rev_page
GROUP BY rev_actor
ORDER BY cou DESC;

```

## Explanation of SQ8

This is an **Aggregation on a Filtered Multi-Range Search**.

### 1. Inner Logic (Search):

- **Relational Filter:** Selects only pages shorter than {len}.

- **Vector Multi-Range Filter:** Selects pages where the semantic distance to the query vector falls into *any* of several specific bands (e.g., "Distance is 0.1-0.2 OR 0.4-0.5").
2. **Outer Logic (Aggregation):**
- Joins the matching pages to the `revision` table.
  - Counts the number of revisions made by each author (`rev_actor`) on those specific pages.

## Indexed Translation: Not Possible

We **cannot** use the vector index (`VECTOR_SEARCH`) for this query.

- **Reason 1 (Range Search):** The index requires a `TOP_N` limit. It cannot find "all items within distance ranges."
- **Reason 2 (Complex Logic):** The index cannot natively handle the `OR` logic between multiple disjoint distance ranges.
- **Reason 3 (Pre-filter):** The index cannot apply the `page_len < {len}` filter *before* traversing the graph.

We must use `VECTOR_DISTANCE` in the `WHERE` clause with complex boolean logic, forcing a full table scan.

## SQ8 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @query_vector VECTOR(384);
```

-- Range 1

```
DECLARE @d1_min FLOAT = 0.1;
DECLARE @d1_max FLOAT = 0.2;
```

-- Range 2

```
DECLARE @d2_min FLOAT = 0.4;
DECLARE @d2_max FLOAT = 0.5;
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
```

```
ORDER BY page_id;
```

```
PRINT '--- Running SQ8: Aggregation on Filtered Multi-Range Search (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT
```

```
    r.rev_user_text AS rev_actor,  
    COUNT(*) AS cou
```

```
FROM
```

```
    dbo.page AS p
```

```
JOIN
```

```
    dbo.revision AS r ON p.page_id = r.rev_page
```

```
WHERE
```

```
-- Relational Filter
```

```
p.page_len < @page_len_limit
```

```
AND (
```

```
-- Multi-Range Vector Logic
```

```
(VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @d1_min
```

```
AND @d1_max)
```

```
OR
```

```
(VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @d2_min
```

```
AND @d2_max)
```

```
)
```

```
GROUP BY
```

```
    r.rev_user_text
```

```
ORDER BY
```

```
    cou DESC;
```

```
GO
```

	rev_actor	cou
1	5	4007
2	7	2457
3	38	1373
4	34	836
5	30	584
6	53	570
7	24	515
8	31	444
9	8	401
10	20	379
11	123	283
12	650	234
13	51	233
14	324	231
15	247	188
16	56	160
17	27	156
18	130	150
19	9	125
~	~	~

Q34 SQ9:

```
SELECT year, old_id, distance
```

```
FROM (
```

```

SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}' AS
distance, ROW_NUMBER() OVER (
    PARTITION BY EXTRACT (YEAR FROM rev_timestamp)
    ORDER BY text_embedding {op} '{q}', old_id
) AS rank FROM text JOIN revision ON old_id = rev_id
WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH}
) AS ranked_pages
WHERE rank in ({r_1},{r_2},...,{r_n})
ORDER BY year DESC, distance ASC;

```

## Explanation of SQ9

This is a **Partitioned Specific Rank Selection**.

- **Logic:**
  1. **Filter:** Selects revisions within a specific year range (`{YEARL}` to `{YEARH}`).
  2. **Partition:** Groups the results by **year**.
  3. **Rank:** Within each year, ranks the text segments by their semantic similarity to the query vector.
  4. **Select:** Picks only specific ranks from each year (e.g., "Get the 1st, 5th, and 10th best match for 2010, for 2011, etc.").
- **Use Case:** Sampling search quality at specific depths across different time periods.

## Indexed Translation: Not Possible

We **cannot** use the vector index (`VECTOR_SEARCH`) for this query.

- **Reason 1 (Partitioning):** The index finds the **global** top N matches. It cannot find "Top N per Year."
- **Reason 2 (Specific Ranks):** The index returns a contiguous block of results (1 to N). It cannot "skip" to fetch only specific ranks (like 5 and 10) without fetching 1-4 and 6-9.

We must use `VECTOR_DISTANCE` inside a window function (`ROW_NUMBER()`), which requires calculating the distance for all rows matching the date filter (a table scan).

## SQ9 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
```

GO

```
-- 1. Define parameters
```

```
DECLARE @year_low INT = 2010; -- {YEARL}
```

```
DECLARE @year_high INT = 2015; -- {YEARH}
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding
```

```
FROM dbo.text
```

```
WHERE text_embedding IS NOT NULL
```

```
ORDER BY old_id;
```

```
PRINT '--- Running SQ9: Partitioned Specific Rank Selection (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT
```

```
ranked_pages.[year],
```

```
ranked_pages.old_id,
```

```
ranked_pages.distance
```

```
FROM (
```

```
SELECT
```

```
t.old_id,
```

```
LEFT(r.rev_timestamp, 4) AS [year], -- Robust Year Extraction
```

```
VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance,
```

```
-- Partition by Year and Rank by Distance

ROW_NUMBER() OVER (
    PARTITION BY LEFT(r.rev_timestamp, 4)
    ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC, t.old_id
    ASC
) AS rank

FROM
dbo.text AS t

JOIN
dbo.revision AS r ON t.old_id = r.rev_text_id

WHERE
-- Filter date range first
CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
) AS ranked_pages

WHERE
-- Filter for specific ranks {r_1, r_2...}
ranked_pages.rank IN (1, 5, 10, 20)

ORDER BY
ranked_pages.[year] DESC,
ranked_pages.distance ASC;

GO
```

	year	old_id	distance
1	2015	22176	0.0394490361213684
2	2015	37596	0.0505063533782959
3	2015	50207	0.0539038181304932
4	2015	907	0.0572460293769836
5	2014	7183	0.0459926724433899
6	2014	16524	0.0555241107940674
7	2014	71992	0.06090633850977
8	2014	99400	0.0649563074111938
9	2013	40853	0.0473921298980713
10	2013	82155	0.0599041500091553
11	2013	91654	0.059153336524963
12	2013	51846	0.0623485249519348
13	2012	18446	0.0415235161781311
14	2012	32693	0.0503955483436584
15	2012	93833	0.0544382333755493
16	2012	85355	0.0610643625259399
17	2011	49406	0.0460441708564758
18	2011	41023	0.0543234944343567
19	2011	46663	0.0574133992195129

Query executed successfully.

(Local) (17.0 RC1) MSI\vedan (81) index\_hybench\_100k 00:00:00 24 rows

### Q35 SQ 10:

```

SELECT year, old_id, distance
FROM (
    SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}' AS distance,
    FROM text JOIN revision ON old_id = rev_id
    WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH}
    AND ((text_embedding {op} '{q}' BETWEEN {d_1} AND {d_1*})
    OR (text_embedding {op} '{q}' BETWEEN {d_2} AND {d_2*})
    OR .. OR (text_embedding {op} '{q}' BETWEEN {d_n} AND {d_n*}))
) AS ranked_pages
ORDER BY year DESC, distance ASC;

```

### Explanation of SQ10

This is a **Filtered Multi-Range Search on a Join**.

- **Logic:**
  1. **Join:** Links the `text` and `revision` tables.
  2. **Filter 1 (Relational):** Selects text revisions within a specific year range (`{YEARL}` to `{YEARH}`).
  3. **Filter 2 (Vector Multi-Range):** Selects text segments where the semantic distance to the query vector falls into *any* of several specific bands (e.g., "0.1-0.2 OR 0.4-0.5").
  4. **Sorting:** Orders the results by year and then by similarity distance.

### Indexed Translation: Not Possible

We **cannot** use the vector index (VECTOR\_SEARCH) for this query.

- **Reason 1 (Range Search):** The index requires a TOP\_N limit. It cannot find "all items within distance ranges."
- **Reason 2 (Complex Logic):** The index cannot natively handle the OR logic between multiple disjoint distance ranges.

We must use VECTOR\_DISTANCE in the WHERE clause, forcing a full table scan.

## SQ10 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @year_low INT = 2010; -- {YEARL}
DECLARE @year_high INT = 2015; -- {YEARH}
DECLARE @query_vector VECTOR(384);
```

-- Range 1

```
DECLARE @d1_min FLOAT = 0.1;
DECLARE @d1_max FLOAT = 0.2;
```

-- Range 2

```
DECLARE @d2_min FLOAT = 0.4;
DECLARE @d2_max FLOAT = 0.5;
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = text_embedding
FROM dbo.text
WHERE text_embedding IS NOT NULL
ORDER BY old_id;
```

```
PRINT '--- Running SQ10: Filtered Multi-Range Search (Slow). ---';
```

-- 3. Run the query

```
SELECT
    LEFT(r.rev_timestamp, 4) AS [year], -- Robust Year Extraction
    t.old_id,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
```

```
-- Filter 1: Date Range
CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high

AND (
    -- Filter 2: Vector Range 1
    (VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d1_min AND
@d1_max)
    OR
    -- Filter 3: Vector Range 2
    (VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d2_min AND
@d2_max)
)
ORDER BY
[year] DESC,
distance ASC;
GO
```

# Indexable

Based on the query translations we completed, we were able to utilize the vector index for **7 queries**.

Here is the breakdown:

## The "Indexable" 7 (Fast Queries)

These queries worked because they relied on **Top-N (k-NN)** logic without strict pre-filtering or distance thresholds.

1. **NQ1**: Standard k-NN Search.
2. **NQ11**: Aggregation on k-NN Search.
3. **NQ13**: Aggregation on k-NN Search (Text table).
4. **NQ15**: Centroid Search (using the Python-SQL hybrid approach).
5. **NQ16**: Multi-Target k-NN (using UNION of two searches).
6. **NQ18**: Exclusion k-NN (using nested searches).
7. **IQ1**: Paginated k-NN (using TOP\_N + SQL OFFSET).

## The "Non-Indexable" Majority (~33 Queries)

The remaining queries failed to use the index due to three specific architectural limitations in the SQL Server 2025 Preview:

1. **Range Searches**: Queries asking for "all items within distance d" (e.g., Q2, Q6, IQ2, SQ2) cannot use VECTOR\_SEARCH because it mandates a fixed TOP\_N limit.
2. **Pre-Filters**: Queries requiring a filter *before* ranking (e.g., Q3, Q5, IQ3) fail because the index acts as a Post-filter, which destroys result accuracy/recall.
3. **Specific Ranks**: Queries asking for non-contiguous ranks (e.g., SQ1 "Rank 50") cannot leverage the index's graph traversal efficiently.

**Result for Report:** You were able to optimize ~17.5% of the workload using the vector index. The remaining 82.5% required full table scans due to current software limitations.

Recall SQLServer'

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

Query Category	Query IDs	Index Used ?	Technical Justification
Standard k-NN	Q1, IQ1	YES	Directly maps to <b>VECTOR_SEARCH</b> with <b>TOP_N</b> .
Agg. on k-NN	NQ11, NQ13	YES	The core operation is a Top-N search; aggregations happen after retrieval.
Multi-Target	NQ16, NQ18	YES	Solved using <b>UNION</b> or nested <b>VECTOR_SEARCH</b> calls to combine multiple Top-N result sets.

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

<b>Centroid</b>	NQ15	<b>YES</b>	Solved using a Hybrid approach (Python computes centroid -> SQL runs Top-N search).
<b>Range Search</b>	Q2, Q6, NQ12, NQ14	<b>NO</b>	The Index API ( <b>VECTOR_SEARCH</b> ) mandates a <b>TOP_N</b> limit. It cannot process unbounded radius searches (e.g., "All items within 0.5 distance").
<b>Pre-Filtered k-NN</b>	Q3, Q5, Q7	<b>NO</b>	The index performs <b>Post-filtering</b> . Using it for Pre-filters (e.g., <code>page_len &lt; 1000</code> ) would discard valid matches early, destroying result accuracy. Correctness requires a full scan.

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

<b>Comple x Range</b>	SQ2, SQ4, SQ6, SQ8	<b>NO</b>	These involve multiple disjoint distance ranges ( <b>OR</b> logic) or bounded bands (annulus search), which the Index API does not support.
<b>Specific Rank</b>	SQ1, SQ3, SQ5, SQ9	<b>NO</b>	The index cannot "skip" to retrieve specific non-contiguous ranks (e.g., "Rank 50") without retrieving ranks 1-49 first.

NQ 11

```
USE index_hybench_100k;
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
GO
SET NOCOUNT ON;

-- 1. Setup K values
DECLARE @K_Values TABLE (K INT);
INSERT INTO @K_Values VALUES (10), (100), (200), (500), (1000);

DECLARE @MaxK INT;
SELECT @MaxK = MAX(K) FROM @K_Values;

-- 2. Pick random query vector from PAGE table
DECLARE @query_vector VECTOR(384);
SELECT TOP 1 @query_vector = page_embedding FROM dbo.page ORDER BY NEWID();

PRINT '-----';
PRINT 'Benchmarking NQ11 (Agg on k-NN) | Recall based on Page Retrieval Accuracy';
PRINT '-----';

-----  

-- STEP A: Run Ground Truth ONCE for @MaxK
-- We find the "True Top K" Pages. If we find the right pages, the aggregation is correct.
-----  

PRINT 'Running Ground Truth Scan...';
DECLARE @gt_start DATETIME2 = SYSUTCDATETIME();

DROP TABLE IF EXISTS #All_GT_IDs;
CREATE TABLE #All_GT_IDs (id INT PRIMARY KEY, rank INT);
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
INSERT INTO #All_GT_IDs (id, rank)
SELECT
    page_id,
    ROW_NUMBER() OVER (ORDER BY VECTOR_DISTANCE('cosine', @query_vector,
page_embedding) ASC)
FROM dbo.page
ORDER BY VECTOR_DISTANCE('cosine', @query_vector, page_embedding) ASC
OFFSET 0 ROWS FETCH NEXT @MaxK ROWS ONLY;

DECLARE @gt_end DATETIME2 = SYSUTCDATETIME();
DECLARE @full_scan_time_ms INT = DATEDIFF(MILLISECOND, @gt_start, @gt_end);

PRINT 'Ground Truth calculated in ' + CAST(@full_scan_time_ms AS VARCHAR(10)) + ' ms.';
PRINT '-----';
PRINT 'K | Recall (%) | Index Time (ms) | Accel. (Est.)';
PRINT '-----';

-----
-- STEP B: Loop
-----

DECLARE @k INT;
DECLARE @cur CURSOR;
SET @cur = CURSOR FOR SELECT K FROM @K_Values ORDER BY K;
OPEN @cur;
FETCH NEXT FROM @cur INTO @k;
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
WHILE @@FETCH_STATUS = 0
BEGIN
    -- 1. Run FULL INDEXED AGGREGATION (To measure realistic latency)
    DECLARE @idx_start DATETIME2 = SYSUTCDATETIME();

    -- We discard the actual aggregation result, we just want the runtime
    DECLARE @dummy INT;
    SELECT @dummy = COUNT(*) FROM (
        SELECT
            LEFT(r.rev_timestamp, 4) AS [year],
            COUNT(*) AS [count]
        FROM
            VECTOR_SEARCH(
                TABLE = dbo.page AS p,
                COLUMN = page_embedding,
                SIMILAR_TO = @query_vector,
                METRIC = 'cosine',
                TOP_N = @k
            ) AS s
        JOIN
            dbo.revision AS r ON p.page_id = r.rev_page
        GROUP BY
            LEFT(r.rev_timestamp, 4)
    ) x;

    DECLARE @idx_end DATETIME2 = SYSUTCDATETIME();
    DECLARE @idx_time_ms INT = DATEDIFF(MILLISECOND, @idx_start, @idx_end);
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

-- 2. Run JUST SEARCH to capture IDs for Recall Calculation

```
DECLARE @Index_IDs TABLE (id INT);
DELETE FROM @Index_IDs;
```

```
INSERT INTO @Index_IDs (id)
SELECT page_id
FROM VECTOR_SEARCH(
    TABLE = dbo.page,
    COLUMN = page_embedding,
    SIMILAR_TO = @query_vector,
    METRIC = 'cosine',
    TOP_N = @k
);
```

-- 3. COMPUTE RECALL

```
DECLARE @overlap INT;
SELECT @overlap = COUNT(*)
FROM @Index_IDs i
WHERE EXISTS (SELECT 1 FROM #All_GT_IDs gt WHERE gt.id = i.id AND gt.rank <= @k);
```

```
DECLARE @recall DECIMAL(5,2) = (CAST(@overlap AS FLOAT) / CAST(@k AS FLOAT)) *
100.0;
```

-- Acceleration calculation

```
DECLARE @acceleration DECIMAL(10,1) = 0;
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
IF @idx_time_ms > 0 SET @acceleration = CAST(@full_scan_time_ms AS FLOAT) /  
CAST(@idx_time_ms AS FLOAT);
```

```
PRINT CAST(@k AS VARCHAR(5)) +  
    CHAR(9) + ' | ' + CAST(@recall AS VARCHAR(10)) + '%' +  
    CHAR(9) + ' | ' + CAST(@idx_time_ms AS VARCHAR(10)) + ' ms' +  
    CHAR(9) + ' | ' + CAST(@acceleration AS VARCHAR(10)) + 'x';
```

```
FETCH NEXT FROM @cur INTO @k;  
END
```

```
CLOSE @cur;  
DEALLOCATE @cur;  
DROP TABLE #All_GT_IDs;
```

---

Benchmarking NQ11 (Agg on k-NN) | Recall based on Page Retrieval Accuracy

---

Running Ground Truth Scan...  
Ground Truth calculated in 1270 ms.

---

K	Recall (%)	Index Time (ms)	Accel. (Est.)
---	------------	-----------------	---------------

---

10	100.00%	69 ms	18.4x
100	100.00%	182 ms	7.0x
200	99.50%	198 mxxxxxxs	6.4x

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

500	99.80%	322 ms	3.9x
1000	99.90%	413 ms	3.1x

## NQ13

```
USE index_hybench_100k;
GO
SET NOCOUNT ON;

-- 1. Setup K values
DECLARE @K_Values TABLE (K INT);
INSERT INTO @K_Values VALUES (10), (100), (200), (500), (1000);

DECLARE @MaxK INT;
SELECT @MaxK = MAX(K) FROM @K_Values;

-- 2. Pick random query vector from TEXT table
DECLARE @query_vector VECTOR(384);
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY NEWID();

PRINT '-----';
PRINT 'Benchmarking NQ13 (Agg on k-NN) | Recall based on Retrieval Accuracy';
PRINT '-----';
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
-- STEP A: Run Ground Truth ONCE for @MaxK
```

```
-- We identify the "True Top K" Text segments.
```

---

```
PRINT 'Running Ground Truth Scan...';
DECLARE @gt_start DATETIME2 = SYSUTCDATETIME();

DROP TABLE IF EXISTS #All_GT_IDs;
CREATE TABLE #All_GT_IDs (id INT PRIMARY KEY, rank INT);

INSERT INTO #All_GT_IDs (id, rank)
SELECT
    old_id,
    ROW_NUMBER() OVER (ORDER BY VECTOR_DISTANCE('cosine', @query_vector,
text_embedding) ASC)
FROM dbo.text
ORDER BY VECTOR_DISTANCE('cosine', @query_vector, text_embedding) ASC
OFFSET 0 ROWS FETCH NEXT @MaxK ROWS ONLY;

DECLARE @gt_end DATETIME2 = SYSUTCDATETIME();
DECLARE @full_scan_time_ms INT = DATEDIFF(MILLISECOND, @gt_start, @gt_end);

PRINT 'Ground Truth calculated in ' + CAST(@full_scan_time_ms AS VARCHAR(10)) + ' ms.';
```

---

```
PRINT '-----';
```

```
PRINT 'K | Recall (%) | Index Time (ms) | Accel. (Est.)';
```

---

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

-- STEP B: Loop

---

```
DECLARE @k INT;
DECLARE @cur CURSOR;
SET @cur = CURSOR FOR SELECT K FROM @K_Values ORDER BY K;
OPEN @cur;
FETCH NEXT FROM @cur INTO @k;

WHILE @@FETCH_STATUS = 0
BEGIN
    -- 1. Run FULL INDEXED AGGREGATION (To measure realistic latency)
    DECLARE @idx_start DATETIME2 = SYSUTCDATETIME();

    -- We discard the result, we just want the runtime
    DECLARE @dummy INT;
    SELECT @dummy = COUNT(*) FROM (
        SELECT
            r.rev_user_text AS rev_actor,
            SUM(CAST(r.rev_minor_edit AS INT)) AS total_minor_edits
        FROM
        (
            SELECT old_id
            FROM VECTOR_SEARCH(
                TABLE = dbo.text AS t,
                COLUMN = text_embedding,
                SIMILAR_TO = @query_vector,
                METRIC = 'cosine',
                K = @k
            )
        ) r
    );

```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
TOP_N = @k
)
) AS new_text
JOIN
    dbo.revision AS r ON new_text.old_id = r.rev_id
GROUP BY
    r.rev_user_text
) x;

DECLARE @idx_end DATETIME2 = SYSUTCDATETIME();
DECLARE @idx_time_ms INT = DATEDIFF(MILLISECOND, @idx_start, @idx_end);

-- 2. Run JUST SEARCH to capture IDs for Recall Calculation
DECLARE @Index_IDs TABLE (id INT);
DELETE FROM @Index_IDs;

INSERT INTO @Index_IDs (id)
SELECT old_id
FROM VECTOR_SEARCH(
    TABLE = dbo.text,
    COLUMN = text_embedding,
    SIMILAR_TO = @query_vector,
    METRIC = 'cosine',
    TOP_N = @k
);
-- 3. COMPUTE RECALL
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
DECLARE @overlap INT;
SELECT @overlap = COUNT(*)
FROM @Index_IDs i
WHERE EXISTS (SELECT 1 FROM #All_GT_IDs gt WHERE gt.id = i.id AND gt.rank <= @k);

DECLARE @recall DECIMAL(5,2) = (CAST(@overlap AS FLOAT) / CAST(@k AS FLOAT)) *
100.0;

-- Acceleration calculation
DECLARE @acceleration DECIMAL(10,1) = 0;
IF @idx_time_ms > 0 SET @acceleration = CAST(@full_scan_time_ms AS FLOAT) /
CAST(@idx_time_ms AS FLOAT);

PRINT CAST(@k AS VARCHAR(5)) +
CHAR(9) + ' | ' + CAST(@recall AS VARCHAR(10)) + '%' +
CHAR(9) + ' | ' + CAST(@idx_time_ms AS VARCHAR(10)) + ' ms' +
CHAR(9) + ' | ' + CAST(@acceleration AS VARCHAR(10)) + 'x';

FETCH NEXT FROM @cur INTO @k;
END

CLOSE @cur;
DEALLOCATE @cur;
DROP TABLE #All_GT_IDs;
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
  2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
  3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
  4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
  5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.
- 

Running Ground Truth Scan...

Ground Truth calculated in 837 ms.

---

K	Recall (%)	Index Time (ms)	Accel. (Est.)
---	------------	-----------------	---------------

---

10	100.00%	16 ms	52.3x
100	100.00%	62 ms	13.5x
200	100.00%	118 ms	7.1x
500	100.00%	277 ms	3.0x
1000	100.00%	407 ms	2.1x

---

## NQ16

```
USE index_hybench_100k;
GO
SET NOCOUNT ON;

-- 1. Setup K values
DECLARE @K_Values TABLE (K INT);
INSERT INTO @K_Values VALUES (10), (100), (200), (500), (1000);

DECLARE @MaxK INT;
SELECT @MaxK = MAX(K) FROM @K_Values;
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

-- 2. Pick TWO random query vectors

```
DECLARE @query_vector_1 VECTOR(384);
DECLARE @query_vector_2 VECTOR(384);
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY NEWID();
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY NEWID();

PRINT '-----';
PRINT 'Benchmarking NQ16 (Multi-Target k-NN) | LEAST(dist1, dist2)';
PRINT '-----';
```

-- STEP A: Run Ground Truth ONCE for @MaxK

-- We calculate the exact LEAST distance for every row and pick the true top K.

```
-----
```

```
PRINT 'Running Ground Truth Scan...';
DECLARE @gt_start DATETIME2 = SYSUTCDATETIME();
```

```
DROP TABLE IF EXISTS #All_GT_IDs;
CREATE TABLE #All_GT_IDs (id INT PRIMARY KEY, rank INT);
```

```
INSERT INTO #All_GT_IDs (id, rank)
SELECT
    old_id,
    ROW_NUMBER() OVER (
        ORDER BY LEAST(
            VECTOR_DISTANCE('cosine', @query_vector_1, text_embedding),
            VECTOR_DISTANCE('cosine', @query_vector_2, text_embedding))
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
    ) ASC
)
FROM dbo.text
ORDER BY LEAST(
    VECTOR_DISTANCE('cosine', @query_vector_1, text_embedding),
    VECTOR_DISTANCE('cosine', @query_vector_2, text_embedding)
) ASC
OFFSET 0 ROWS FETCH NEXT @MaxK ROWS ONLY;

DECLARE @gt_end DATETIME2 = SYSUTCDATETIME();
DECLARE @full_scan_time_ms INT = DATEDIFF(MILLISECOND, @gt_start, @gt_end);

PRINT 'Ground Truth calculated in ' + CAST(@full_scan_time_ms AS VARCHAR(10)) + ' ms.';
PRINT '-----';
PRINT 'K    | Recall (%) | Index Time (ms) | Accel. (Est.)';
PRINT '-----';

-----
-- STEP B: Loop
-----

DECLARE @k INT;
DECLARE @cur CURSOR;
SET @cur = CURSOR FOR SELECT K FROM @K_Values ORDER BY K;
OPEN @cur;
FETCH NEXT FROM @cur INTO @k;

WHILE @@FETCH_STATUS = 0
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
BEGIN
    -- 1. Run FULL INDEXED QUERY (To measure realistic latency)
    DECLARE @idx_start DATETIME2 = SYSUTCDATETIME();

    -- We discard the result, we just want the runtime
    DECLARE @dummy INT;
    SELECT @dummy = COUNT(*) FROM (
        SELECT TOP (@k)
            t.old_id,
            t.old_text,
            merged.best_distance
        FROM (
            -- Inner Logic: Combine and Deduplicate IDs
            SELECT
                id_source.old_id,
                MIN(id_source.dist) as best_distance
            FROM (
                SELECT t1.old_id, s1.distance as dist
                FROM VECTOR_SEARCH(TABLE = dbo.text AS t1, COLUMN = text_embedding,
SIMILAR_TO = @query_vector_1, METRIC = 'cosine', TOP_N = @k) AS s1

                UNION ALL

                SELECT t2.old_id, s2.distance as dist
                FROM VECTOR_SEARCH(TABLE = dbo.text AS t2, COLUMN = text_embedding,
SIMILAR_TO = @query_vector_2, METRIC = 'cosine', TOP_N = @k) AS s2
            ) AS id_source
    )
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
        GROUP BY id_source.old_id
    ) AS merged
    JOIN dbo.text AS t ON merged.old_id = t.old_id -- Join back for text at the end
    ORDER BY merged.best_distance ASC
) X;

DECLARE @idx_end DATETIME2 = SYSUTCDATETIME();
DECLARE @idx_time_ms INT = DATEDIFF(MILLISECOND, @idx_start, @idx_end);

-- 2. Run JUST SEARCH to capture IDs for Recall
DECLARE @Index_IDs TABLE (id INT);
DELETE FROM @Index_IDs;

INSERT INTO @Index_IDs (id)
SELECT TOP (@k) old_id
FROM (
    SELECT t1.old_id, s1.distance
    FROM VECTOR_SEARCH(TABLE = dbo.text AS t1, COLUMN = text_embedding,
SIMILAR_TO = @query_vector_1, METRIC = 'cosine', TOP_N = @k) AS s1
    UNION ALL
    SELECT t2.old_id, s2.distance
    FROM VECTOR_SEARCH(TABLE = dbo.text AS t2, COLUMN = text_embedding,
SIMILAR_TO = @query_vector_2, METRIC = 'cosine', TOP_N = @k) AS s2
) AS merged
ORDER BY distance ASC;

-- 3. COMPUTE RECALL
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
DECLARE @overlap INT;
SELECT @overlap = COUNT(*)
FROM (SELECT DISTINCT id FROM @Index_IDs) i -- Ensure we count distinct IDs
WHERE EXISTS (SELECT 1 FROM #All_GT_IDs gt WHERE gt.id = i.id AND gt.rank <= @k);

DECLARE @recall DECIMAL(5,2) = (CAST(@overlap AS FLOAT) / CAST(@k AS FLOAT)) *
100.0;

-- Acceleration calculation
DECLARE @acceleration DECIMAL(10,1) = 0;
IF @idx_time_ms > 0 SET @acceleration = CAST(@full_scan_time_ms AS FLOAT) /
CAST(@idx_time_ms AS FLOAT);

PRINT CAST(@k AS VARCHAR(5)) +
CHAR(9) + '|' + CAST(@recall AS VARCHAR(10)) + '%' +
CHAR(9) + '|' + CAST(@idx_time_ms AS VARCHAR(10)) + ' ms' +
CHAR(9) + '|' + CAST(@acceleration AS VARCHAR(10)) + 'x';

FETCH NEXT FROM @cur INTO @k;
END

CLOSE @cur;
DEALLOCATE @cur;
DROP TABLE #All_GT_IDs;
```

Benchmarking NQ16 (Multi-Target k-NN) | LEAST(dist1, dist2)

---

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

Running Ground Truth Scan...

Ground Truth calculated in 835 ms.

---

K	Recall (%)	Index Time (ms)	Accel. (Est.)
---	------------	-----------------	---------------

---

10	100.00%	21 ms	39.8x
100	100.00%	131 ms	6.4x
200	100.00%	121 ms	6.9x
500	99.20%	246 ms	3.4x
1000	99.50%	395 ms	2.1x
10000	92.73%	2527 ms	0.3x

NQ18

```
USE index_hybench_100k;
GO
SET NOCOUNT ON;

-- 1. Setup K values
DECLARE @K_Values TABLE (K INT);
INSERT INTO @K_Values VALUES (10), (100), (200),(500), (1000);

-- 2. Pick TWO random query vectors
DECLARE @query_vector_1 VECTOR(384);
DECLARE @query_vector_2 VECTOR(384);
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY NEWID();
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY NEWID();

PRINT '-----';
PRINT 'Benchmarking NQ18 (k-NN with Exclusion) | Recall & Latency';
PRINT '-----';
PRINT 'K    | Recall (%) | Index Time (ms) | Exact Time (ms) | Acceleration';
PRINT '-----';

DECLARE @k INT;
DECLARE @cur CURSOR;
SET @cur = CURSOR FOR SELECT K FROM @K_Values ORDER BY K;
OPEN @cur;
FETCH NEXT FROM @cur INTO @k;

WHILE @@FETCH_STATUS = 0
BEGIN
    DECLARE @k_buffer INT = @k * 2; -- Fetch double to ensure we have enough after
    exclusion

    -----
    -- A. GROUND TRUTH (Exact Search - Two Full Scans)
    -- Logic: Find Top K close to Q1, excluding Top K close to Q2
    -----

    DECLARE @gt_start DATETIME2 = SYSUTCDATETIME();

    DECLARE @GT_IDs TABLE (id INT PRIMARY KEY);
    DELETE FROM @GT_IDs;
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
INSERT INTO @GT_IDs (id)
SELECT TOP (@k) t.old_id
FROM dbo.text AS t
WHERE t.old_id NOT IN (
    -- Inner Scan: Exclusion List (Top K closest to Q2)
    SELECT TOP (@k) old_id
    FROM dbo.text
    ORDER BY VECTOR_DISTANCE('cosine', @query_vector_2, text_embedding) ASC
)
ORDER BY VECTOR_DISTANCE('cosine', @query_vector_1, t.text_embedding) ASC;

DECLARE @gt_end DATETIME2 = SYSUTCDATETIME();
DECLARE @gt_time_ms INT = DATEDIFF(MILLISECOND, @gt_start, @gt_end);

-----
-- B. INDEX SEARCH (Nested Vector Search)
-----

DECLARE @idx_start DATETIME2 = SYSUTCDATETIME();

DECLARE @Index_IDs TABLE (id INT PRIMARY KEY);
DELETE FROM @Index_IDs;

INSERT INTO @Index_IDs (id)
SELECT TOP (@k) t.old_id
FROM
    -- Outer Search: Find candidates for Q1 (with buffer)
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
VECTOR_SEARCH(
    TABLE = dbo.text AS t,
    COLUMN = text_embedding,
    SIMILAR_TO = @query_vector_1,
    METRIC = 'cosine',
    TOP_N = @k_buffer
) AS s1
WHERE
    t.old_id NOT IN (
        -- Inner Search: Find forbidden IDs for Q2
        SELECT t2.old_id
        FROM VECTOR_SEARCH(
            TABLE = dbo.text AS t2,
            COLUMN = text_embedding,
            SIMILAR_TO = @query_vector_2,
            METRIC = 'cosine',
            TOP_N = @k
        ) AS s2
    )
ORDER BY
    s1.distance ASC;

DECLARE @idx_end DATETIME2 = SYSUTCDATETIME();
DECLARE @idx_time_ms INT = DATEDIFF(MILLISECOND, @idx_start, @idx_end);

-----
-- C. COMPUTE RECALL
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
-----  
DECLARE @overlap INT;  
SELECT @overlap = COUNT(*)  
FROM @Index_IDs i  
WHERE EXISTS (SELECT 1 FROM @GT_IDs g WHERE g.id = i.id);  
  
DECLARE @recall DECIMAL(5,2) = (CAST(@overlap AS FLOAT) / CAST(@k AS FLOAT)) *  
100.0;  
  
DECLARE @acceleration DECIMAL(10,1) = 0;  
IF @idx_time_ms > 0 SET @acceleration = CAST(@gt_time_ms AS FLOAT) /  
CAST(@idx_time_ms AS FLOAT);  
  
PRINT CAST(@k AS VARCHAR(5)) +  
CHAR(9) + ' | ' + CAST(@recall AS VARCHAR(10)) + '%' +  
CHAR(9) + ' | ' + CAST(@idx_time_ms AS VARCHAR(10)) + ' ms' +  
CHAR(9) + ' | ' + CAST(@gt_time_ms AS VARCHAR(10)) + ' ms' +  
CHAR(9) + ' | ' + CAST(@acceleration AS VARCHAR(10)) + 'x';  
  
FETCH NEXT FROM @cur INTO @k;  
END  
  
CLOSE @cur;  
DEALLOCATE @cur;  
PRINT '-----';
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

---

#### Benchmarking NQ18 (k-NN with Exclusion) | Recall & Latency

---

K	Recall (%)	Index Time (ms)	Exact Time (ms)	Acceleration
---	------------	-----------------	-----------------	--------------

---

10	20.00%	221 ms	1199 ms	5.4x
100	2.00%	4471 ms	2332 ms	0.5x
200	2.50%	15602 ms	1764 ms	0.1x
500	1.80%	65823 ms	1399 ms	0.0x
1000	1.30%	221751 ms	1284 ms	0.0x

---

IQ1:

```
USE index_hybench_100k;
```

```
GO
```

```
SET NOCOUNT ON;
```

```
-- 1. Setup K values (We treat K as the "End Rank" of the page we are fetching)
```

```
DECLARE @K_Values TABLE (K INT);
```

```
INSERT INTO @K_Values VALUES (10), (100), (200), (500), (1000);
```

```
DECLARE @MaxK INT;
```

```
SELECT @MaxK = MAX(K) FROM @K_Values;
```

```
-- 2. Pick random query vector
```

```
DECLARE @query_vector VECTOR(384);
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY NEWID();
```

```
PRINT '-----';  
PRINT 'Benchmarking IQ1 (Paginated k-NN) | Fetching last 10 items ending at Rank K';  
PRINT '-----';
```

```
-----  
-- STEP A: Run Ground Truth ONCE for @MaxK  
-- We calculate the exact ranking for the top 1000 items.  
-----
```

```
PRINT 'Running Ground Truth Scan...';  
DECLARE @gt_start DATETIME2 = SYSUTCDATETIME();
```

```
DROP TABLE IF EXISTS #All_GT_IDs;  
CREATE TABLE #All_GT_IDs (id INT PRIMARY KEY, rank INT);
```

```
INSERT INTO #All_GT_IDs (id, rank)  
SELECT  
    old_id,  
    ROW_NUMBER() OVER (ORDER BY VECTOR_DISTANCE('cosine', @query_vector,  
    text_embedding) ASC)  
FROM dbo.text  
ORDER BY VECTOR_DISTANCE('cosine', @query_vector, text_embedding) ASC  
OFFSET 0 ROWS FETCH NEXT @MaxK ROWS ONLY;
```

```
DECLARE @gt_end DATETIME2 = SYSUTCDATETIME();  
DECLARE @full_scan_time_ms INT = DATEDIFF(MILLISECOND, @gt_start, @gt_end);
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
PRINT 'Ground Truth calculated in ' + CAST(@full_scan_time_ms AS VARCHAR(10)) + ' ms.';  
PRINT '-----';  
PRINT 'End Rank(r)| Recall (%) | Index Time (ms) | Accel. (Est.)';  
PRINT '-----';  
  
-----  
-- STEP B: Loop  
-----  
DECLARE @k INT; -- This acts as 'r' (End Rank)  
DECLARE @cur CURSOR;  
SET @cur = CURSOR FOR SELECT K FROM @K_Values ORDER BY K;  
OPEN @cur;  
FETCH NEXT FROM @cur INTO @k;  
  
WHILE @@FETCH_STATUS = 0  
BEGIN  
    -- Pagination Logic: Fetch a page of size 10 ending at @k  
    DECLARE @page_size INT = 10;  
    DECLARE @r INT = @k;  
    DECLARE @l INT = @r - @page_size + 1; -- Start Rank  
  
    DECLARE @offset INT = @l - 1;  
    DECLARE @limit INT = @page_size; -- {r-l+1}  
    DECLARE @fetch_total INT = @r; -- Index must fetch up to r  
  
    -- 1. Run INDEXED PAGINATION Query
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
DECLARE @idx_start DATETIME2 = SYSUTCDATETIME();

DECLARE @Index_IDs TABLE (id INT);
DELETE FROM @Index_IDs;

INSERT INTO @Index_IDs (id)
SELECT
    t.old_id
FROM
    VECTOR_SEARCH(
        TABLE = dbo.text AS t,
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector,
        METRIC = 'cosine',
        TOP_N = @fetch_total -- Must fetch top r
    ) AS s
ORDER BY
    s.distance ASC,
    t.old_id ASC
-- Apply Pagination
OFFSET @offset ROWS
FETCH NEXT @limit ROWS ONLY;

DECLARE @idx_end DATETIME2 = SYSUTCDATETIME();
DECLARE @idx_time_ms INT = DATEDIFF(MILLISECOND, @idx_start, @idx_end);

-- 2. COMPUTE RECALL
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
-- Check if the retrieved IDs match the GT IDs for the specific ranks l to r
DECLARE @overlap INT;
SELECT @overlap = COUNT(*)
FROM @Index_IDs i
WHERE EXISTS (
    SELECT 1
    FROM #All_GT_IDs gt
    WHERE gt.id = i.id
    AND gt.rank BETWEEN @l AND @r -- Check against specific rank slice
);
-- Recall is based on the page size (10), not @k
DECLARE @recall DECIMAL(5,2) = (CAST(@overlap AS FLOAT) / CAST(@page_size AS
FLOAT)) * 100.0;

DECLARE @acceleration DECIMAL(10,1) = 0;
IF @idx_time_ms > 0 SET @acceleration = CAST(@full_scan_time_ms AS FLOAT) /
CAST(@idx_time_ms AS FLOAT);

PRINT CAST(@k AS VARCHAR(5)) +
    CHAR(9) + ' | ' + CAST(@recall AS VARCHAR(10)) + '%' +
    CHAR(9) + ' | ' + CAST(@idx_time_ms AS VARCHAR(10)) + ' ms' +
    CHAR(9) + ' | ' + CAST(@acceleration AS VARCHAR(10)) + 'x';

FETCH NEXT FROM @cur INTO @k;
END
```

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

```
CLOSE @cur;
DEALLOCATE @cur;
DROP TABLE #All_GT_IDs;
```

---

Benchmarking IQ1 (Paginated k-NN) | Fetching last 10 items ending at Rank K

---

Running Ground Truth Scan...

Ground Truth calculated in 1832 ms.

---

End Rank(r)| Recall (%) | Index Time (ms) | Accel. (Est.)

---

10	90.00%	166 ms	11.0x
100	80.00%	376 ms	4.9x
200	60.00%	331 ms	5.5x
500	0.00%	670 ms	2.7x
1000	0.00%	760 ms	2.4x

Completion time: 2025-11-28T14:50:08.6542729+05:30

This behavior is expected for **Paginated Search** on vector indexes, and the "0% recall" at depth (e.g., ranks 491–500) is due to a specific phenomenon called **Ranking Instability**.

#### 1. Why Recall Drops to 0% at Depth

## Q1 (NQ1)

Calculating Recall@K directly in SQL Server is the most accurate way to verify the index performance because it keeps the data movement inside the database engine.

Since we are doing this "one by one," we will start with **Query 1 (Q1)**, which is the standard k-Nearest Neighbor search.

### Benchmark Script for Q1 (Standard k-NN)

This T-SQL script does the following:

1. Picks a random **Query Vector** from your table.
2. Loops through your desired **K values** (10, 100, 200, 500, 1000).
3. Runs the **Exact Search** (Slow, Ground Truth) and records the IDs.
4. Runs the **Index Search** (Fast, **VECTOR\_SEARCH**) and records the IDs.
5. Compares the two sets to calculate **Recall** (accuracy) and prints the **Latency** (time taken) for both.

Your benchmark is testing **Slice Recall** (e.g., "Do the 10 items at ranks 991–1000 in the Index match the exact 10 items at ranks 991–1000 in Ground Truth?").

This is an extremely strict test that fails for two reasons:

1. **The "Dense Cluster" Problem:** In high-dimensional vector spaces (384-dim), you often have hundreds of items with almost identical distances (e.g., **0.4561** vs **0.4562**).
  - **Ground Truth:** Might sort Item A as rank 490 and Item B as rank 491.
  - **Index (Approximate):** Might calculate slightly different float approximations and sort Item B as 490 and Item A as 491.
  - **Result:** If your page boundary is at 490, the index pushes the item *just outside* your requested page. When this happens for all 10 items in a narrow slice, you get 0% overlap for that specific page, even if the index found 99% of the correct items *overall*.
2. **ANN Degradation:** **DiskANN** is optimized to find the "Top N" quickly. As N gets larger (e.g., 1000), the "greedy" pathfinding algorithm has to explore deeper into the graph. It becomes less accurate at distinguishing the "999th best" item from the "1001st best" item.

**Recommendation for Report:** Clarify that while **Recall@1000** (the whole set) might be high (e.g., 95%), **Precision-at-Rank-1000** (the specific order of the tail) is inherently unstable in approximate nearest neighbor search.

# Formal Proofs

Here are the formal proofs for the four categories of non-indexable queries, formatted in LaTeX.

I have provided the **rendered view** first so you can check the logic, followed by the **Raw LaTeX Source Code** at the bottom that you can copy and paste directly into your Overleaf or LaTeX document.

---

## 1. Bucket A: Unbounded Range Searches

- **Queries:** Q2, Q4, Q6, NQ12, NQ14, IQ2, IQ4, IQ6, IQ8, IQ10, SQ2, SQ4, SQ6, SQ8, SQ10.
- **Definition:** Queries requesting all vectors within a semantic distance threshold.

**Formal Proof of Incompatibility:**

**Premise 1:** The SQL Server `VECTOR_SEARCH` function strictly requires a `TOP_N` parameter ( $\tau \in \mathbb{Z}^+$ ) to define the search termination condition.

Premise 2: The result set  $R$  of a range query is defined as:

$$R = \{x \in DB \mid \text{dist}(x, q) < \delta\}$$

The cardinality  $|R|$  is non-deterministic prior to execution and lies in the range  $[0, |DB|]$ .

**Logical Contradiction:** To utilize `VECTOR_SEARCH`, a specific  $\tau$  must be selected.

1. **Case  $\tau < |R|$ :** The index returns a subset of valid results, resulting in **Incorrect Recall**.
2. **Case  $\tau \geq |DB|$ :** To guarantee correctness without knowing  $|R|$ , one must set  $\tau$  to the maximum table size. This forces the index to traverse the entire graph structure, degrading the operation complexity to  $O(N)$  or worse due to random memory access patterns, rendering it computationally inferior to a sequential Full Table Scan.

**Conclusion:** Therefore, an unbounded range search cannot be efficiently executed using a Top-N constrained index architecture.

---

## 2. Bucket B: Pre-Filtered k-NN Searches

- **Queries:** Q3, Q5, Q7, IQ3, IQ5, IQ7, SQ3, SQ7.
- **Definition:** Queries requiring a relational filter to be applied *before* ranking.

**Formal Proof of Incompatibility:**

**Premise 1:** The `VECTOR_SEARCH` function executes the Approximate Nearest Neighbor (ANN) search on the vector graph  $\$G\$$  independently and prior to external relational filtering (Post-Filtering architecture).

Premise 2: Let  $S_{\text{global}}$  be the set of the top  $k$  nearest neighbors in the entire database:

$\$S_{\text{global}} = \text{\textbackslash text\{k-NN\}(q, DB)}$

Let  $S_{\text{filtered}}$  be the set of rows satisfying the relational predicate  $P$ :

$\$S_{\text{filtered}} = \{x \mid P(x) \text{ is true}\}$

Scenario: Consider a distribution where the semantic nearest neighbors do not satisfy the relational predicate:

$\$S_{\text{global}} \cap S_{\text{filtered}} = \emptyset$

#### Execution Trace:

1. The Index retrieves  $S_{\text{global}}$ .
2. The Query Engine applies predicate  $P$  to  $S_{\text{global}}$ .
3. The final result set  $R' = \{x \mid P(x)\} = \emptyset$ .

**Contradiction:** The query requirement is to return  $k$  items from  $S_{\text{filtered}}$ . Provided  $|S_{\text{filtered}}| \geq k$ , the correct result set is non-empty. Thus, the Post-Filtering index yields an incorrect result (0 items instead of  $k$ ).

**Conclusion:** Therefore, Pre-Filtered k-NN logic is logically incompatible with a Post-Filtering index architecture.

---

### 3. Bucket C: Specific Rank Selection

- **Queries:** SQ1, SQ3, SQ5, SQ9.
- **Definition:** Queries requesting non-contiguous specific ranks (e.g., Rank 50).

#### Formal Proof of Incompatibility:

**Premise 1:** The DiskANN graph traversal algorithm is sequential regarding proximity. To discover the node at Rank  $R_n$ , the algorithm must traverse the graph path through neighbors  $R_1 \dots R_{n-1}$ .

**Premise 2:** The `VECTOR_SEARCH` API returns a contiguous result set  $[1, \tau]$ . It does not support a sparse selection operator.

#### Inefficiency Proof:

Let the query request a set of specific ranks  $Q_{\text{ranks}} = \{r_1, r_2, \dots, r_m\}$  where  $r_m$  is the maximum requested rank.

To satisfy this, the index parameter must be set to  $\tau = r_m$ .

The Data Retrieval Overhead ( $O_{\text{data}}$ ) is defined as:

$$O_{\text{data}} = \frac{\text{Records Retrieved}}{\text{Records Requested}} = \frac{r_m}{m}$$
  
Example: For a query requesting only  $\{100\}$  (\$ $m=1$ ,  $r_m=100$ ):

$$O_{\text{data}} = \frac{100}{1} = 100$$

This indicates fetching 99% unnecessary data payload.

**Conclusion:** Using the index imposes a prohibitive I/O overhead for sparse rank retrieval.

---

## 4. Bucket D: Dissimilarity / Minimax Logic

- **Queries:** NQ17, NQ19.
- **Definition:** Queries where sorting depends on multiple query vectors simultaneously.

### Formal Proof of Incompatibility:

**Premise 1:** The vector index construction optimizes the graph  $G$  for minimizing the distance metric  $\text{dist}(x, q)$  to a single query point  $q$  within the Voronoi tessellation of the vector space.

Premise 2: The Minimax sorting function  $F(x)$  is defined as:

$$F(x) = \max(\text{dist}(x, q_1), \text{dist}(x, q_2))$$

Execution Failure:

The decision boundary for minimizing  $F(x)$  does not map linearly to the proximity graph of either  $q_1$  or  $q_2$  individually.

- An item  $x^*$  may minimize  $F(x)$  by being equidistant to  $q_1$  and  $q_2$ .
- However,  $x^*$  may be distinctively far from both  $q_1$  (Rank  $> k$ ) and  $q_2$  (Rank  $> k$ ).
- Therefore,  $x^* \notin \text{k-NN}(q_1, DB) \cup \text{k-NN}(q_2, DB)$ .

**Conclusion:** Since the optimal item for the composite function is not guaranteed to appear in the top- $k$  neighborhood of any single query vector, a single-vector Nearest Neighbor index cannot guarantee retrieval.

---

## Raw LaTeX Source Code

Copy the block below into your report file.

Code snippet

- `\section{Formal Proofs of Non-Indexability}`
- 
- The following sections formally define why specific HyBench query categories could not utilize the SQL Server 2025 `\texttt{DiskANN}` vector index.
- 
- `\subsection{Bucket A: Unbounded Range Searches}`
- `\textbf{Definition:}` Queries requesting all vectors where the semantic distance is below a threshold  $\delta$  (e.g., Q2, Q6, NQ12).
- 
- `\begin{proof}`
- `\textbf{Premise 1:}` The SQL Server `\texttt{VECTOR\_SEARCH}` function strictly requires a `\texttt{TOP_N}` parameter ( $\tau \in \mathbb{Z}^+$ ) to define the search termination condition.
- 
- `\textbf{Premise 2:}` The result set  $R$  of a range query is defined as:
- $$R = \{x \in DB \mid \text{dist}(x, q) < \delta\}$$
- The cardinality  $|R|$  is non-deterministic prior to execution and lies in the range  $[0, |DB|]$ .
- 
- `\textbf{Logical Contradiction:}` To utilize the index, a specific  $\tau$  must be selected.
- `\begin{enumerate}`
- `\item \textbf{Case } \tau < |R|:` The index returns a truncated subset of valid results, resulting in `\textit{Incorrect Recall}`.
- `\item \textbf{Case } \tau \geq |DB|:` To guarantee correctness without knowing  $|R|$ , one must set  $\tau \approx |DB|$ . This forces the index to traverse the entire graph structure, degrading the operation complexity to  $O(N)$  or worse due to random memory access patterns, rendering it computationally inferior to a sequential Full Table Scan.
- `\end{enumerate}`
- `\textbf{Conclusion:}` An unbounded range search cannot be efficiently executed using a Top-N constrained index architecture.
- `\end{proof}`
- 
- `\subsection{Bucket B: Pre-Filtered k-NN Searches}`
- `\textbf{Definition:}` Queries requiring a relational filter to be applied before ranking (e.g., Q3, Q5, IQ3).
- 
- `\begin{proof}`
- `\textbf{Premise 1:}` The `\texttt{VECTOR\_SEARCH}` function executes the Approximate Nearest Neighbor (ANN) search on the vector graph  $G$  independently and prior to external relational filtering (Post-Filtering architecture).
- 
- `\textbf{Premise 2:}` Let  $S_{\text{global}}$  be the set of the top  $k$  nearest neighbors in the entire database:
- $$S_{\text{global}} = \text{dist}(q, DB)$$
- Let  $S_{\text{filtered}}$  be the set of rows satisfying the relational predicate  $P$ :
- $$S_{\text{filtered}} = \{x \in DB \mid P(x) \text{ is true}\}$$

- \textbf{Scenario:} Consider a distribution where the semantic nearest neighbors do not satisfy the relational predicate:  

$$\exists S_{\text{global}} \cap S_{\text{filtered}} = \emptyset$$
- \textbf{Execution Trace:}
- \begin{enumerate}
- \item The Index retrieves  $S_{\text{global}}$ .
- \item The Query Engine applies predicate  $P$  to  $S_{\text{global}}$ .
- \item The final result set  $R' = \{x \in S_{\text{global}} \mid P(x)\} = \emptyset$ .
- \end{enumerate}
- \textbf{Contradiction:} The query logic requires returning  $k$  items from  $S_{\text{filtered}}$ . Provided  $|S_{\text{filtered}}| \geq k$ , the correct result set is non-empty. Thus, the Post-Filtering index yields an incorrect result (0 items instead of  $k$ ).
- \textbf{Conclusion:} Pre-Filtered k-NN logic is logically incompatible with a Post-Filtering index architecture.
- \end{proof}
- \subsection{Bucket C: Specific Rank Selection}
- \textbf{Definition:} Queries requesting non-contiguous specific ranks (e.g., SQ1 requesting Rank 50 and 100).
- \begin{proof}
- \textbf{Premise 1:} The DiskANN graph traversal algorithm is sequential regarding proximity. To discover the node at Rank  $R_n$ , the algorithm must traverse the graph path through neighbors  $R_1 \dots R_{n-1}$ .
- \textbf{Premise 2:} The \texttt{VECTOR\\_SEARCH} API returns a contiguous result set  $[1, \tau]$ . It does not support a sparse selection operator.
- \textbf{Inefficiency Proof:}
- Let the query request a set of specific ranks  $Q_{\text{ranks}} = \{r_1, r_2, \dots, r_m\}$  where  $r_m$  is the maximum requested rank.
- To satisfy this, the index parameter must be set to  $\tau = r_m$ .
- The Data Retrieval Overhead ( $O_{\text{data}}$ ) is defined as:  

$$O_{\text{data}} = \frac{\text{Records Retrieved}}{\text{Records Requested}} = \frac{r_m}{m}$$
- \textbf{Example:} For a query requesting only  $\{ \text{Rank}_{100} \}$  ( $m=1, r_m=100$ ):  

$$O_{\text{data}} = \frac{100}{1} = 100$$
- This indicates fetching 99% unnecessary data payload.
- \textbf{Conclusion:} Using the index imposes a prohibitive I/O overhead for sparse rank retrieval compared to the selective nature of the query.

- `\end{proof}`
- 
- `\subsection{Bucket D: Dissimilarity / Minimax Logic}`
- `\textbf{Definition:}` Queries where sorting depends on multiple query vectors simultaneously (e.g., NQ17).
- 
- `\begin{proof}`
- `\textbf{Premise 1:}` The vector index construction optimizes the graph  $G$  for minimizing the distance metric  $\text{dist}(x, q)$  to a `\textit{single}` query point  $q$  within the Voronoi tessellation of the vector space.
- 
- `\textbf{Premise 2:}` The Minimax sorting function  $F(x)$  is defined as:
- $$\boxed{F(x) = \max(\text{dist}(x, q_1), \text{dist}(x, q_2))}$$
- 
- `\textbf{Execution Failure:}`
- The decision boundary for minimizing  $F(x)$  does not map linearly to the proximity graph of either  $q_1$  or  $q_2$  individually.
- `\begin{itemize}`
- `\item An item  $x^*$  may minimize  $F(x)$  by being equidistant to  $q_1$  and  $q_2$ .`
- `\item However,  $x^*$  may be distinctively far from both  $q_1$  (Rank  $> k$ ) and  $q_2$  (Rank  $> k$ ).`
- `\item Therefore,  $x^* \notin \text{k-NN}(q_1, DB) \cup \text{k-NN}(q_2, DB)$ .`
- `\end{itemize}`
- 
- `\textbf{Conclusion:}` Since the optimal item for the composite function is not guaranteed to appear in the top- $k$  neighborhood of any single query vector, a single-vector Nearest Neighbor index cannot guarantee retrieval.
- `\end{proof}`

-

**Tab 8**

```
PS C:\Users\mukes\Desktop\postgres> python run_all.py
>>
```

Found 38 SQL files.

Running in alphabetical order:

- Running Q1(NQ1).sql ...
  - ✓ Completed in 3090.068 ms
- Running Q10(NQ12).sql ...
  - ✓ Completed in 1292.887 ms
- Running Q11(NQ13).sql ...
  - ✓ Completed in 848.352 ms
- Running Q12(NQ14).sql ...
  - ✓ Completed in 783.148 ms
- Running Q13(NQ9).sql ...
  - ✓ Completed in 331.976 ms
- Running Q14(NQ10).sql ...
  - ✓ Completed in 222.031 ms
- Running Q16(IQ1).sql ...
  - ✓ Completed in 749.999 ms
- Running Q17(IQ2).sql ...
  - ✓ Completed in 609.568 ms
- Running Q18(IQ3).sql ...
  - ✓ Completed in 194.875 ms
- Running Q19(IQ4).sql ...
  - ✓ Completed in 252.433 ms
- Running Q2(NQ2).sql ...
  - ✓ Completed in 534.812 ms
- Running Q20(IQ5).sql ...
  - ✓ Completed in 229.182 ms
- Running Q21(IQ6).sql ...
  - ✓ Completed in 177.179 ms
- Running Q22(IQ7).sql ...
  - ✓ Completed in 220.067 ms
- Running Q23(IQ8).sql ...
  - ✓ Completed in 244.623 ms
- Running Q24(IQ9).sql ...
  - ✓ Completed in 463.988 ms
- Running Q25(IQ10).sql ...
  - ✓ Completed in 408.076 ms
- Running Q26(SQ1).sql ...
  - ✓ Completed in 940.401 ms
- Running Q27(SQ2).sql ...
  - ✓ Completed in 2089.904 ms

- Running Q28(SQ3).sql ...
  - ✓ Completed in 363.501 ms
- Running Q29(SQ4).sql ...
  - ✓ Completed in 249.184 ms
- Running Q3(NQ3).sql ...
  - ✓ Completed in 194.107 ms
- Running Q30(SQ5).sql ...
  - ✓ Completed in 541.653 ms
- Running Q31(SQ6).sql ...
  - ✓ Completed in 377.292 ms
- Running Q32(SQ7).sql ...
  - ✓ Completed in 294.269 ms
- Running Q33(SQ8).sql ...
  - ✓ Completed in 251.27 ms
- Running Q34(SQ9).sql ...
  - ✓ Completed in 178.237 ms
- Running Q35(SQ10).sql ...
  - ✓ Completed in 3079.605 ms
- Running Q36(NQ16).sql ...
  - ✓ Completed in 2868.522 ms
- Running Q37(NQ18).sql ...
  - ✓ Completed in 1762.716 ms
- Running Q38(NQ19).sql ...
  - ✓ Completed in 1033.181 ms
- Running Q39(NQ17).sql ...
  - ✓ Completed in 1229.138 ms
- Running Q4(NQ4).sql ...
  - ✓ Completed in 202.228 ms
- Running Q5(NQ5).sql ...
  - ✓ Completed in 227.082 ms
- Running Q6(NQ6).sql ...
  - ✓ Completed in 213.758 ms
- Running Q7(NQ7).sql ...
  - ✓ Completed in 216.416 ms
- Running Q8(NQ8).sql ...
  - ✓ Completed in 287.199 ms
- Running Q9(NQ11).sql ...
  - ✓ Completed in 209.373 ms

=====

ALL QUERIES COMPLETE!

Results saved to latency\_results.csv

=====

PS C:\Users\mukes\Desktop\postgres>

# NQ - Postgres

## Q1(NQ1)

```
SELECT old_id,old_text
FROM text
ORDER BY text_embedding {op} '{q}',old_id
LIMIT {k};
```

## SQLServer

```
USE index_hybench_500k_new;
GO
```

```
-- 1. Define our K value and query vector
DECLARE @k INT = 10;
DECLARE @query_vector VECTOR(384); -- Declare as the correct VECTOR type
```

```
-- 2. Grab a real vector from the table to search with
-- (No CAST to VARCHAR needed, we can select the vector directly)
```

```
SELECT TOP 1
    @query_vector = text_embedding
FROM
    dbo.text
WHERE
    text_embedding IS NOT NULL
ORDER BY
    old_id; -- Just to get a consistent vector each time
```

```
PRINT '--- Running a k-NN search (k=10) using VECTOR_DISTANCE. ---';
PRINT '--- This will be VERY SLOW (1-2+ minutes) as it is a full table scan. ---';
```

```
-- 3. Run the k-NN search
```

```
-- This is the correct "no-index" translation
```

```
SELECT TOP (@k) -- This is the T-SQL equivalent of LIMIT {k}
    t.old_id,
    t.old_text,
    -- This is the corrected function call with 3 arguments
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
ORDER BY
    distance ASC, -- Primary sort: ORDER BY text_embedding {op} '{q}'
    t.old_id ASC; -- Secondary sort (tie-breaker)
```

GO

	old_id	old_text	distance
1	1	<strong>MediaWiki has been installed.</strong>\n\n...	0
2	367891	Name: Marcus M. Edvall\n\nCompany: [http://tomo...	0.0223326683044434
3	107909	Health forums refers to a [http://en.wikipedia.org/wik...	0.0277445316314697
4	135968	I am ""SaintHermit".\nI don't wanna be a sim...	0.0288935899734497
5	201240	[https://www.deviantart.com/kazumikuchi Devianta...	0.0291732549667358
6	80021	" <b>Hello!</b> \n\nThank you for contributions. I h...	0.0298174023628235
7	315742	I am an entrepreneur and software developer.\n\n...	0.0309765338897705
8	216418	Karen Morrione\n\n==My Interests==\n\n==S...	0.0327359437942505
9	446771	Links solicited (and pending):\n* [http://www.zoosav...	0.0327522754669189
10	337087	-- Noneluder.com ==\nLaunched on February 1...	0.0328952670097351

✓ Que... | MUKESH\MSSQLSERVER01 (17.0 ... | MUKESH\mukes (64) | index\_hybench\_500k | 00:00:13 | 10 rows

## PostgreSQLQ1(NQ1)

-- Step 1: Pick the same first vector as SQL Server

```
WITH q AS (
    SELECT text_embedding AS query_vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
)
```

-- Step 2: Perform full table scan k-NN using cosine (<=>)

```
SELECT
    t.old_id,
    t.old_text,
    (t.text_embedding <=> q.query_vec) AS distance -- cosine distance
FROM
    text t, q
WHERE
    t.text_embedding IS NOT NULL
ORDER BY
    distance ASC,
    t.old_id ASC
LIMIT 10; -- same as @k
```

	old_id [PK] integer	old_text text	distance double precision
1	1	<strong>MediaWiki has been installed.</strong>\n\nConsult t...	0
2	367891	Name: Marcus M. Edvall\n\nCompany: [http://tomopt.com/ To...	0.022332716691471632
3	107909	Health forums refers to a [http://en.wikipedia.org/wiki/Bulletin_b...	0.027744552173902703
4	135968	I am "SaintHermit". \nI don't wanna be a simple hermit, I ...	0.028893573986948584
5	201240	[https://www.deviantart.com/kazumikikuchi Deviantart]   [https:/...	0.029173281957399766
6	80021	"<b> Hello! </b>"\n\nThank you for contributions. I have realiz...	0.029817488918771007
7	315742	I am an entrepreneur and software developer.\n\nI have made ...	0.03097655527876242
8	216418	Karen Morrione\n\n==My Interests==\n\n==Subpages...	0.03273601285048988
9	446771	Links solicited (and pending):\n* [http://www.zoosavvy.com Ani...	0.032752297660286867
10	337087	-- Noneluder.com ==\n\nLaunched on February 1st, 2008, [htt...	0.03289538984080287

Total rows: 10

Query complete 00:00:15.221

## Q2(NQ2)

```
SELECT old_id,old_text
FROM text
WHERE text_embedding {op} '{q}' < {d}
ORDER BY text_embedding {op} '{q}',old_id;
```

SQLServer

```
USE index_hybench_500k;
GO
```

```
-- 1. Define our distance threshold and query vector
DECLARE @distance_threshold FLOAT = 0.05; -- Example distance. {d}
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector from the table to search with
SELECT TOP 1
    @query_vector = text_embedding
```

```

FROM
    dbo.text
WHERE
    text_embedding IS NOT NULL
ORDER BY
    old_id;

```

PRINT '--- Running a range search (distance < 0.5). This will be VERY SLOW.  
---';

```

-- 3. Run the range search query
-- This translates the Postgres query
SELECT
    t.old_id,
    t.old_text,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
WHERE
    -- This is the translation of: text_embedding {op} '{q}' < {d}
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) <
    @distance_threshold
ORDER BY
    distance ASC, -- Primary sort: ORDER BY text_embedding {op} '{q}'
    t.old_id ASC; -- Secondary sort (tie-breaker)
GO

```

	old_id	old_text	distance
1	1	<strong>MediaWiki has been installed.</strong>\n...\n	0
2	367891	Name: Marcus M. Edval\n\nCompany: [http://tomo...	0.0223326683044434
3	107909	Health forums refers to a [http://en.wikipedia.org/wik...	0.0277445316314697
4	135968	I am ""SaintHermit"". \nI don't wanna be a sim...	0.0288935899734497
5	201240	[https://www.deviantart.com/kazumikuchi Devianta...	0.0291732549667358
6	80021	"<b> Hello! </b>"\n\nThank you for contributions. I h...	0.0298174023628235
7	315742	I am an entrepreneur and software developer.\n\n...	0.0309765338897705
8	216418	Karen Morrione\n\n==My Interests==\n\n\n==S...	0.0327359437942505
9	446771	Links solicited (and pending):\n* [http://www.zoosav...	0.0327522754669189
10	337087	== Noneluder.com ==\n\nLaunched on February 1...	0.0328952670097351

✓ Qu | MUKESH\MSSQLSERVER01 (17.0 ... | MUKESH\mukes (64) | index hvbench 500k | 00:00:05 | 5.510 rows

## PostgreSQLQ2(NQ2)

```
-- Step 1: Select the same query vector as SQL Server (first vector by old_id)
```

```
WITH q AS (
```

```
    SELECT text_embedding AS query_vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
)
```

```
-- Step 2: Range search (distance < d) using cosine distance (<=>)
```

```
SELECT
```

```
    t.old_id,
    t.old_text,
    (t.text_embedding <=> q.query_vec) AS distance
FROM
```

```
    text t, q
WHERE
```

```
    (t.text_embedding <=> q.query_vec) < 0.05 -- your {d}
```

```
ORDER BY
```

```
    distance ASC,
    t.old_id ASC;
```

	old_id [PK] integer	old_text text	distance double precision
1	1	<strong>MediaWiki has been installed.</strong>\n\nConsult t...	0
2	367891	Name: Marcus M. Edvall\n\nCompany: [http://tomopt.com/ To...	0.022332716691471632
3	107909	Health forums refers to a [http://en.wikipedia.org/wiki/Bulletin_b...	0.027744552173902703
4	135968	I am "SaintHermit". \nI don't wanna be a simple hermit, I ...	0.028893573986948584
5	201240	[https://www.deviantart.com/kazumikuchi Deviantart]   [https:/...	0.029173281957399766
6	80021	"<b> Hello! </b>"\n\nThank you for contributions. I have realiz...	0.029817488918771007
7	315742	I am an entrepreneur and software developer.\n\nI have made ...	0.03097655527876242
8	216418	Karen Morrione\n\n==My Interests==\n\n==Subpages...	0.03273601285048988
9	446771	Links solicited (and pending):\n* [http://www.zoosavvy.com Ani...	0.032752297660286867
10	337087	== Noneluder.com ==\n\nLaunched on February 1st, 2008, [htt...	0.03289538984080287
11	186867	==Useful links==\n[[tools:~dispenser/view/Checklinks Toolserv...	0.032988091876017145
Total rows: 5510		Query complete 00:00:14.355	CRLF Ln 22, Col 1

## Query 3 (Q3) Translation

This query adds a standard `WHERE` clause. In T-SQL, this is a "pre-filter" that happens before the vector sort. Like our other queries on `HyBenchDB`, it will be slow because it must perform a full table scan, but it is the correct translation.

```
SELECT page_id,page_title  
FROM page  
WHERE page_len < {len}  
ORDER BY page_embedding {op} '{q}',page_id  
LIMIT {k};
```

SQLServer

```
USE index_hybench_500k;  
GO  
  
-- 1. Enable preview features (needed for VECTOR_SEARCH)  
ALTER DATABASE SCOPED CONFIGURATION  
SET PREVIEW_FEATURES = ON;  
GO  
-- 2. Define parameters  
DECLARE @k INT = 10;  
DECLARE @page_length_limit INT = 1000;  
DECLARE @query_vector VECTOR(384);  
  
-- 3. Grab a real vector  
SELECT TOP 1 @query_vector = page_embedding  
FROM dbo.page  
WHERE page_embedding IS NOT NULL  
ORDER BY page_id;  
  
PRINT '--- Running Q3 (Fast, Post-Filter) using VECTOR_SEARCH. ---';  
  
-- 4. Run the query  
SELECT  
    t.page_id,
```

```

t.page_title,
s.distance AS cosine_distance
FROM
    VECTOR_SEARCH(
        TABLE = dbo.page AS t,
        COLUMN = page_embedding,
        SIMILAR_TO = @query_vector,
        METRIC = 'cosine',
        TOP_N = @k -- Find top 10 *first*
    ) AS s
WHERE
    t.page_len < @page_length_limit -- Apply filter *after*
ORDER BY
    s.distance ASC,
    t.page_id ASC;
GO

```

	page_id	page_title	cosine_distance
1	86669	Main_effect	0.0298356413841248
2	323695	Open_class	0.0309374928474426
3	458436	Content_format	0.0312483310699463
4	395057	Table_Head	0.0315550565719604
5	325456	Log_House	0.031710684299469
6	361769	Head_piece	0.033022403717041
7	491821	Head_end	0.0331647396087646

✓ Quer... | MUKESH\MSSQLSERVER01 (17.0 ... | MUKESH\mukes (64) | index\_hybench\_500k | 00:00:00 | 7 rows

## PostgreSQL

```

-- Step 1: Pick the same query vector as SQL Server
WITH q AS (
    SELECT page_embedding AS query_vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
)

```

```

-- Step 2: PRE-FILTER → ORDER → LIMIT
SELECT
    t.page_id,

```

```

t.page_title,
(t.page_embedding <= q.query_vec) AS distance
FROM
page t, q
WHERE
t.page_len < 1000
ORDER BY
distance ASC,
t.page_id ASC
LIMIT 10;

```

	page_id [PK] integer	page_title	distance double precision
1	1	Main_Page	0
2	45003	Page_layout	0.027832891262170478
3	323695	Open_class	0.03093746664704633
4	205690	Side_box	0.03103917986053506
5	458436	Content_form...	0.03124837695262206
6	395057	Table_Head	0.031555073979153136
7	325456	Log_House	0.031710618234469634
8	361769	Head_piece	0.03302242665547106
9	491821	Head_end	0.03316465357371945
10	122511	File_system	0.03340714293283775

Total rows: 10    Query complete 00:00:03.201

## Query 4 (NQ4) Translation

```

SELECT page_id,page_title
FROM page
WHERE page_len < {len} AND page_embedding {op} '{q}' < d

```

```
ORDER BY page_embedding {op} '{q}',page_id;
```

### Critical Analysis: Why `VECTOR_SEARCH` Cannot Be Used

This is a key finding for your project.

The `VECTOR_SEARCH` function (which uses the index) is only designed for **k-NN (Top-N) searches**. It *cannot* be used for pure "range searches" (i.e., `WHERE distance < {d}`).

1. **k-NN Search (Fast):** "Give me the 10 rows with the *smallest* distance."
  - o This is what `VECTOR_SEARCH(..., TOP_N = 10)` does.
  - o This is what Q1 and Q3 (Translation 3A) are.
2. **Range Search (Slow):** "Give me *all* rows where the distance is *less than 0.5*."
  - o This is what Q2 and Q4 are.
  - o `VECTOR_SEARCH` has no parameter for this.
  - o Therefore, you **must** use `VECTOR_DISTANCE` in the `WHERE` clause.

**Conclusion:** Because this query must use `VECTOR_DISTANCE` in the `WHERE` clause, SQL Server is forced to perform a **full table scan**. It will calculate the distance for every single row in the table, check if it passes the filter, and *then* sort the results.

This means **this query will not use your index**, and it will be just as slow on your 100k-row indexed database as it is on your 200k-row non-indexed database (relative to their size). This is a crucial limitation to note in your evaluation.

SQLServer

```
USE index_hybench_500k_new; -- GO
```

```
-- 1. Define parameters
```

```
DECLARE @distance_threshold FLOAT = 0.5; -- Example for {d}
```

```
DECLARE @page_length_limit INT = 1000; -- Example for {len}
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector from the page table
```

```
SELECT TOP 1
```

```
@query_vector = page_embedding  
FROM  
    dbo.page  
WHERE  
    page_embedding IS NOT NULL  
ORDER BY  
    page_id;
```

```
PRINT '--- Running Q4: Hybrid Filter (page_len < 1000 AND distance < 0.5). ---';  
PRINT '--- This will be VERY SLOW (full table scan) on ALL databases. ---';
```

```
-- 3. Run the query
```

```
SELECT  
    p.page_id,  
    p.page_title,  
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) AS distance  
FROM  
    dbo.page AS p  
WHERE  
    -- This is the translation of the hybrid WHERE clause  
    p.page_len < @page_length_limit  
    AND  
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) < @distance_threshold  
ORDER BY
```

```
distance ASC, -- ORDER BY page_embedding {op} '{q}'  
p.page_id ASC; -- Tie-breaker  
  
GO
```

PostgreSQL

```
-- Step 1: Select the same query vector as SQL Server  
WITH q AS (  
    SELECT page_embedding AS query_vec  
    FROM page  
    WHERE page_embedding IS NOT NULL  
    ORDER BY page_id  
    LIMIT 1  
)  
  
-- Step 2: Hybrid filter (page_len < 1000 AND distance < 0.5)  
SELECT  
    p.page_id,  
    p.page_title,  
    (p.page_embedding <=> q.query_vec) AS distance  
FROM  
    page p,  
    q  
WHERE  
    p.page_len < 1000  
    AND (p.page_embedding <=> q.query_vec) < 0.5  
ORDER BY  
    distance ASC,  
    p.page_id ASC  
LIMIT 10; -- optional (SQL Server uses TOP)
```

	page_id [PK] integer	page_title	distance double precision
1	1	Main_Page	0
2	45003	Page_layout	0.027832891262170478
3	323695	Open_class	0.03093746664704633
4	205690	Side_box	0.03103917986053506
5	458436	Content_form...	0.03124837695262206
6	395057	Table_Head	0.031555073979153136
7	325456	Log_House	0.031710618234469634
8	361769	Head_piece	0.03302242665547106
9	491821	Head_end	0.03316465357371945
10	122511	File_system	0.03340714293283775

Total rows: 10    Query complete 00:00:04.353

Q5(NQ5)

```
SELECT rev_id
FROM text JOIN revision ON old_id=rev_id
WHERE rev_timestamp>='{DATE_LOW}' AND rev_timestamp<='{DATE_HIGH}'
ORDER BY text_embedding {op} '{q}',old_id
LIMIT {k};
```

SQLServer

```
USE index_hybench_500k_new; -- Or HyBenchDB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @k INT = 10;
```

```
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z'; -- Example {DATE_LOW}
```

```
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z'; -- Example {DATE_HIGH}
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text;
```

```
PRINT '--- Running Q5 (Slow, Pre-Filter) using VECTOR_DISTANCE... ---';
```

-- 3. Run the query

```
SELECT TOP (@k) -- LIMIT {k}
    r.rev_id,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id -- Join
WHERE
    r.rev_timestamp >= @date_low -- Pre-filter
    AND r.rev_timestamp <= @date_high
ORDER BY
    distance ASC, -- ORDER BY text_embedding {op} '{q}'
    t.old_id ASC; -- Tie-breaker
GO
```

PostgreSQL

-- Step 1: Pick the same query vector as SQL Server

WITH q AS (

SELECT text\_embedding AS query\_vec

FROM text

WHERE text\_embedding IS NOT NULL

ORDER BY old\_id

LIMIT 1

)

-- Step 2: Pre-filter + distance + ordering + limit

SELECT

r.rev\_id,

(t.text\_embedding <= q.query\_vec) AS distance

FROM

text t

JOIN

revision r ON t.old\_id = r.rev\_text\_id

JOIN

q ON TRUE

WHERE

r.rev\_timestamp >= '2010-01-01T00:00:00Z'

AND r.rev\_timestamp <= '2015-01-01T00:00:00Z'

ORDER BY

```
distance ASC,  
t.old_id ASC  
  
LIMIT 10;
```

	rev_id [PK] integer	distance double precision
1	216418	0.03273601285048988
2	446771	0.032752297660286867
3	186867	0.032988091876017145
4	188321	0.034301004882023034
5	415696	0.034354896615113484
6	399179	0.0357271440198037
7	417845	0.03710060248636404
8	459207	0.03750234892774951
9	461212	0.038115730216859656
10	226024	0.038221664555340706

Total rows: 10    Query complete 00:00:05.056

## Q6(NQ6)

```
SELECT rev_id  
FROM text JOIN revision ON old_id=rev_id  
WHERE text_embedding {op} '{q}'<{d} AND rev_timestamp>='{DATE_LOW}' AND  
rev_timestamp<='{DATE_HIGH}'  
ORDER BY text_embedding {op} '{q}',old_id;
```

This query (Q6) is a **hybrid range search**. It joins the `text` and `revision` tables to find all articles that match **two** filters:

1. A standard SQL filter (the revision timestamp is within a date range).
2. A vector range filter (the article's text vector is closer than a specific distance  $\{d\}$  to the query vector).

Indexed Translation: Not Possible

It is **not possible** to translate this query to use the vector index.

The indexed function, `VECTOR_SEARCH`, is only for k-NN (`TOP_N`) queries. Since this query is a **range search** (it asks for everything *under* a distance threshold `{d}`, not the *top k* results), we cannot use `VECTOR_SEARCH`.

The only correct translation must use the `VECTOR_DISTANCE` function in the `WHERE` clause. This will force a full table scan and **will not** use the index.

SQLServer

Here is the only correct, runnable translation for this query. It will be slow.

```
USE index_hybench_500k_new; -- Or HyBenchDB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @distance_threshold FLOAT = 0.5; -- Example for {d}
```

```
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z';
```

```
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z';
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```
PRINT '--- Running Q6 (Hybrid Range Search). This will be SLOW (full scan). ---';
```

```
-- 3. Run the query
```

```
SELECT
```

```
r.rev_id,
```

```
VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
```

```
FROM
```

```
dbo.text AS t  
JOIN  
dbo.revision AS r ON t.old_id = r.rev_text_id  
  
WHERE  
-- Filter 1: Standard SQL  
r.rev_timestamp >= @date_low  
AND r.rev_timestamp <= @date_high  
-- Filter 2: Vector Range Search  
AND VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) < @distance_threshold
```

ORDER BY

```
distance ASC,  
t.old_id ASC;
```

POstgreSQL

-- Step 1: Choose the same query vector as SQL Server

```
WITH q AS (  
SELECT text_embedding AS query_vec  
FROM text  
WHERE text_embedding IS NOT NULL  
ORDER BY old_id  
LIMIT 1  
)
```

-- Step 2: Hybrid range search (distance < 0.5 AND timestamp in range)

```
SELECT
    r.rev_id,
    (t.text_embedding <=> q.query_vec) AS distance
FROM
    text t
JOIN
    revision r ON t.old_id = r.rev_text_id
JOIN
    q ON TRUE
WHERE
    -- SQL Filter 1: Timestamp range
    r.rev_timestamp >= '2010-01-01T00:00:00Z'
    AND r.rev_timestamp <= '2015-01-01T00:00:00Z'
    -- SQL Filter 2: Vector range search
    AND (t.text_embedding <=> q.query_vec) < 0.5
ORDER BY
    distance ASC,
    t.old_id ASC;
```

	<b>rev_id</b> [PK] integer	<b>distance</b> double precision
1	216418	0.03273601285048988
2	446771	0.032752297660286867
3	186867	0.032988091876017145
4	188321	0.034301004882023034
5	415696	0.034354896615113484
6	399179	0.0357271440198037
7	417845	0.03710060248636404
8	459207	0.03750234892774951
9	461212	0.038115730216859656
10	226024	0.038221664555340706

Total rows: 49918      Query complete 00:00:09.528

## Q7 NQ7

```

SELECT rev_actor,count(*) AS cou
FROM (
    SELECT page_id
    FROM page
    WHERE page_len < {len}
    ORDER BY page_embedding {op} '{q}', page_id
    LIMIT {k}
) AS new_page JOIN revision ON page_id=rev_page
GROUP BY rev_actor
ORDER BY cou desc;

```

## Explanation of Query 7

This is an **Aggregation Query on a Filtered k-NN Search**.

1. **Inner Part:** It performs a k-NN search (`LIMIT {k}`) on the `page` table, but with a **Pre-filter** (`WHERE page_len < {len}`). It wants the top 10 vectors *from the pool of short pages*.
  2. **Outer Part:** It joins those specific pages to the `revision` table to count how many contributions each author (`rev_actor`) made to those specific pages.
- 

Indexed Translation: Not Possible

It is **not possible** to translate this query to use the vector index while preserving the correct logic.

- **Reason:** As we discovered with Q3 and Q5, the indexed function `VECTOR_SEARCH` is a **Post-filter**. It finds the global top `k` similar pages *first*, and only then checks if they are short (`page_len < {len}`).
- **Impact:** If the top `k` globally similar pages happen to be long documents, `VECTOR_SEARCH` would filter them out and return fewer than `k` results (or zero), failing the query's requirement to find exactly `k` pages.

To correctly perform a **Pre-filter** (finding the top `k` *among* the short pages), we must use `VECTOR_DISTANCE` in the `ORDER BY` clause, which forces a full table scan.

SQLServer

```
USE index_hybench_500k_new; -- Use your main 200k DB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @k INT = 10;
```

```
DECLARE @page_len_limit INT = 1000; -- {len}
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector from the page table  
  
SELECT TOP 1 @query_vector = page_embedding  
  
FROM dbo.page  
  
WHERE page_embedding IS NOT NULL  
  
ORDER BY page_id;
```

```
PRINT '--- Running Q7: Aggregation on k-NN with Pre-Filter. ---';  
  
PRINT '--- This will be SLOW (full table scan). ---';
```

```
-- 3. Run the query
```

```
SELECT  
    r.rev_user_text AS rev_actor, -- 'rev_user_text' maps to 'rev_actor' in our schema  
    COUNT(*) AS cou  
  
FROM  
(  
    -- Inner Query: k-NN with Pre-filter (must use VECTOR_DISTANCE)  
    SELECT TOP (@k)  
        p.page_id  
  
    FROM  
        dbo.page AS p  
  
    WHERE  
        p.page_len < @page_len_limit -- Pre-filter  
  
    ORDER BY  
        VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
```

```
p.page_id ASC  
) AS new_page  
JOIN  
dbo.revision AS r ON new_page.page_id = r.rev_page  
GROUP BY  
r.rev_user_text  
ORDER BY  
cou DESC;
```

GO

PostgreSQL

-- Step 1: Select same query vector as SQL Server

```
WITH q AS (  
SELECT page_embedding AS query_vec  
FROM page  
WHERE page_embedding IS NOT NULL  
ORDER BY page_id  
LIMIT 1  
,
```

-- Step 2: Inner k-NN with pre-filter (TOP 10 among short pages)

```
new_page AS (  
SELECT  
p.page_id,  
(p.page_embedding <=> q.query_vec) AS distance
```

```
FROM
page p,
q
WHERE
p.page_len < 1000 -- pre-filter
ORDER BY
distance ASC,
p.page_id ASC
LIMIT 10
)
```

```
-- Step 3: Outer aggregation (count revisions grouped by actor)

SELECT
r.rev_user_text AS rev_actor,
COUNT(*) AS cou
FROM
new_page np
JOIN
revision r ON np.page_id = r.rev_page
GROUP BY
r.rev_user_text
ORDER BY
cou DESC;
```

	rev_actor	cou
	text	bigint
1	7	2
2	1734	1
3	18578	1
4	2	1
5	1717	1
6	3357	1
7	5	1
8	53	1
9	25965	1

Total rows: 9    Query complete 00:00:06.647

## Q8 (NQ8)

```
SELECT rev_actor, count(*) AS cou
FROM page JOIN revision ON page_id=rev_page
WHERE page_embedding {op} '{q}'<{d} AND page_len < {len}
GROUP BY rev_actor
ORDER BY cou DESC;
```

### Explanation

This is an **Aggregation on a Hybrid Range Search**.

1. **Join:** It joins `page` and `revision` to link articles to their authors (`rev_actor`).
2. **Filter 1 (Relational):** It selects only short pages (`page_len < {len}`).

3. **Filter 2 (Vector Range):** It selects only pages that are semantically similar to the query vector within a specific distance threshold (<  $\{d\}$ ).
4. **Aggregation:** It counts how many such pages each author has edited.

Indexed Translation: Not Possible

Just like Q6, this is a **range search** (finding all records within distance  $\{d\}$ ), not a k-NN search (finding the top  $k$  records). The **VECTOR\_SEARCH** function does not support range thresholds, so we **cannot** use the vector index. We must use **VECTOR\_DISTANCE** in the **WHERE** clause.

This will perform a full table scan.

SQLServer

```
USE index_hybench_500k_new; -- Use your 200k DB
GO
```

```
-- 1. Define parameters
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @distance_threshold FLOAT = 0.5; -- {d}
DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running Q9: Aggregation on Hybrid Range Search. ---';
PRINT '--- This will be SLOW (full table scan). ---';
```

```
-- 3. Run the query
SELECT
    r.rev_user_text AS rev_actor, -- Maps to 'rev_actor'
    COUNT(*) AS cou
FROM
    dbo.page AS p
JOIN
    dbo.revision AS r ON p.page_id = r.rev_page
WHERE
    -- Relational Filter
    p.page_len < @page_len_limit
    -- Vector Range Filter
```

```

        AND VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) <
@distance_threshold
GROUP BY
    r.rev_user_text
ORDER BY
    cou DESC;
GO

```

PostgreSQL

```

-- Step 1: Select same query vector as SQL Server
WITH q AS (
    SELECT page_embedding AS query_vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
)

-- Step 2: Hybrid Range Search + Aggregation
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM
    page p
JOIN
    revision r ON p.page_id = r.rev_page
JOIN
    q ON TRUE
WHERE
    -- Relational Filter
    p.page_len < 1000
    -- Vector Range Filter
    AND (p.page_embedding <= q.query_vec) < 0.5
GROUP BY
    r.rev_user_text
ORDER BY
    cou DESC;

```

	rev_actor	cou
	text	bigint
1	8	88372
2	5	32494
3	7	21268
4	38	9850
5	2379	6786
6	34	6149
7	233	4219
8	20	3760
9	31	3325
10	223	2838

Total rows: 31862    Query complete 00:00:08.513

## Q9(NQ11)

```

SELECT EXTRACT (YEAR FROM rev_timestamp) AS year, COUNT(*)
FROM revision JOIN (
    SELECT page_id
    FROM page
    ORDER BY page_embedding {op} '{q}', page_id
    LIMIT {k}
) AS filtered_pages ON revision.rev_page = filtered_pages.page_id
GROUP BY year;

```

Explanation of NQ11 (Q9)

This is an **Aggregation on a k-NN Search**.

- Inner Logic (The Search):** It performs a pure vector similarity search to find the top  $\{k\}$  pages most similar to the query vector  $\{q\}$ .
- Outer Logic (The Aggregation):** It joins those specific top  $\{k\}$  pages to the `revision` table and groups the results by year to show the activity timeline for those topics.

Indexed Translation: Possible 

Yes, this query **can** use the vector index.

The vector search (`ORDER BY ... LIMIT {k}`) is the *first* step and has no pre-filters attached to it. This maps perfectly to `VECTOR_SEARCH`, which retrieves the top matches efficiently using the index before passing them to the standard SQL engine for joining and counting.

SQLServer

```
USE index_hybench_500k_new;
GO
```

```
-- 1. Enable Preview Features
```

```
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;
GO
```

```
-- 2. Define parameters
```

```
DECLARE @k INT = 10;
DECLARE @query_vector VECTOR(384);
```

```
-- 3. Grab a real vector
```

```
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running NQ11 (Fixed): Aggregation on k-NN using String Slicing ---';
```

```
-- 4. Run the query
```

```
SELECT
    -- FIX: Just grab the first 4 chars.
    -- This works for '2024-01-01...', '2024/01/01...', etc.
    LEFT(r.rev_timestamp, 4) AS [year],
    COUNT(*) AS [count]
FROM
    VECTOR_SEARCH(
        TABLE = dbo.page AS p,
        COLUMN = page_embedding,
        SIMILAR_TO = @query_vector,
```

```

    METRIC = 'cosine',
    TOP_N = @k
) AS s
JOIN
    dbo.revision AS r ON p.page_id = r.rev_page
GROUP BY
    LEFT(r.rev_timestamp, 4)
ORDER BY
    [year] DESC;
GO

```

PostgreSQL

```

-- Step 1: Select the same query vector as SQL Server
WITH q AS (
    SELECT page_embedding AS query_vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
),
-- Step 2: Inner k-NN (Top 10 nearest pages)
filtered_pages AS (
    SELECT
        p.page_id,
        (p.page_embedding <=> q.query_vec) AS distance
    FROM
        page p,
        q
    ORDER BY
        distance ASC,
        p.page_id ASC
    LIMIT 10
)
-- Step 3: Aggregate by YEAR
SELECT
    EXTRACT(YEAR FROM r.rev_timestamp::timestamptz) AS year,
    COUNT(*) AS count
FROM
    revision r
JOIN

```

```

filtered_pages fp ON r.rev_page = fp.page_id
GROUP BY
    year
ORDER BY
    year DESC;

```

	year numeric 	count bigint 
1	2024	4
2	2023	1
3	2022	1
4	2017	1
5	2013	1
6	2008	2

Total rows: 6 | Query complete 00:00:06.053

## Q10(NQ12)

```

SELECT EXTRACT (YEAR FROM rev_timestamp) AS year, COUNT(*)
FROM revision JOIN page ON rev_page=page_id
WHERE page_embedding {op} '{q}' < {d}
GROUP BY year;

```

### Explanation of Q10 (NQ12)

This is an **Aggregation on a Vector Range Search**.

- Filtering:** It finds *all* pages that are semantically similar to the query vector within a specific distance threshold ( $< \{d\}$ ). This is a range search, not a "Top K" search.
- Joining & Aggregating:** It joins those pages to the `revision` table and counts the revisions per year.

## Indexed Translation: Not Possible

Since this query uses a **distance threshold** ( $< \{d\}$ ) rather than a limit  $k$ , we **cannot** use the `VECTOR_SEARCH` index function. We must use the `VECTOR_DISTANCE` function in the `WHERE` clause, which forces a full table scan.

SQLServer

```
USE index_hybench_500k_new; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @distance_threshold FLOAT = 0.5; -- {d}
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running NQ12: Aggregation on Range Search (Slow). ---';
```

-- 3. Run the query

```
SELECT
    -- Use the string slicing trick for robust year extraction
    LEFT(r.rev_timestamp, 4) AS [year],
    COUNT(*) AS [count]
FROM
    dbo.page AS p
JOIN
    dbo.revision AS r ON p.page_id = r.rev_page
WHERE
    -- Range Search Filter (Index cannot be used)
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) < @distance_threshold
GROUP BY
    LEFT(r.rev_timestamp, 4)
ORDER BY
    [year] DESC;
GO
```

PostgreSQL

```
-- Step 1: Select same query vector as SQL Server
```

```
WITH q AS (
    SELECT page_embedding AS query_vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
)

-- Step 2: Range Search + Join + Aggregate
SELECT
    EXTRACT(YEAR FROM r.rev_timestamp::timestamptz) AS year,
    COUNT(*) AS count
FROM
    page p
JOIN
    revision r ON p.page_id = r.rev_page
JOIN
    q ON TRUE
WHERE
    (p.page_embedding <=> q.query_vec) < 0.5 -- Range threshold: {d}
GROUP BY
    year
ORDER BY
    year DESC;
```

	year numeric	count bigint
1	2024	146953
2	2023	43594
3	2022	104175
4	2021	15225
5	2020	10024
6	2019	6787
7	2018	6936
8	2017	8078
9	2016	4758
10	2015	4682

Total rows: 23      Query complete 00:00:12.385

## Q11(NQ13)

```
SELECT rev_actor, SUM(rev_minor_edit) AS total_minor_edits
FROM(
    SELECT old_id
    FROM text
    ORDER BY text_embedding {op} '{q}', old_id
    LIMIT {k}
) AS new_text JOIN revision ON old_id=rev_id
GROUP BY rev_actor
ORDER BY total_minor_edits DESC;
```

This is an **Aggregation on a k-NN Search**.

- Inner Logic (The Search):** It performs a pure vector similarity search on the `text` table to find the top `{k}` text segments most similar to the query vector `{q}`.
- Outer Logic (The Aggregation):** It joins those top `{k}` text records to the `revision` table (using `old_id=rev_id`) to find who wrote them (`rev_actor`). It then sums up the number of minor edits made by each actor on these specific texts.

Indexed Translation: Possible 

Yes, this query **can** use the vector index.

The inner query is a standard "Top K" search without any pre-filters. This maps directly to `VECTOR_SEARCH`, which efficiently retrieves the closest matches using the index before the join and aggregation happen.

SQLServer

## NQ13 Translation (Indexed)

This query should be fast on your 100k indexed database.

```
USE index_hybench_500k_new; -- Use your indexed DB  
GO
```

-- 1. Enable Preview Features

```
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;  
GO
```

-- 2. Define parameters

```
DECLARE @k INT = 10;  
DECLARE @query_vector VECTOR(384);
```

-- 3. Grab a real vector

```
SELECT TOP 1 @query_vector = text_embedding  
FROM dbo.text  
WHERE text_embedding IS NOT NULL  
ORDER BY old_id;
```

```
PRINT '--- Running NQ13: Aggregation on k-NN (Indexed). ---';
```

-- 4. Run the query

```
SELECT  
    r.rev_user_text AS rev_actor,  
    SUM(CAST(r.rev_minor_edit AS INT)) AS total_minor_edits -- Cast BIT to INT for SUM  
FROM  
(  
    -- Inner Step: Use Index to find Top K similar texts  
    SELECT old_id  
    FROM VECTOR_SEARCH(  
        TABLE = dbo.text AS t,  
        COLUMN = text_embedding,  
        SIMILAR_TO = @query_vector,  
        METRIC = 'cosine',
```

```

        TOP_N = @k
    )
) AS new_text -- This alias represents the results of VECTOR_SEARCH
JOIN
    dbo.revision AS r ON new_text.old_id = r.rev_id -- Join on text ID
GROUP BY
    r.rev_user_text
ORDER BY
    total_minor_edits DESC;
GO

```

### PostgreSQL

```

-- Step 1: pick the same query vector as SQL Server
WITH q AS (
    SELECT text_embedding AS query_vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
),
-- Step 2: indexed k-NN search (top 10)
topk AS (
    SELECT old_id
    FROM text, q
    ORDER BY text_embedding <=> q.query_vec
    LIMIT 10
)
-- Step 3: aggregate minor edits per actor
SELECT
    r.rev_user_text AS rev_actor,
    SUM(r.rev_minor_edit::int) AS total_minor_edits
FROM
    topk t
JOIN
    revision r ON t.old_id = r.rev_id
GROUP BY
    r.rev_user_text
ORDER BY
    total_minor_edits DESC;

```

	rev_actor	total_minor_edits
	text	bigint
1	34379	1
2	187	1
3	38413	1
4	42117	1
5	17057	0
6	36032	0
7	25395	0
8	19781	0
9	2	0
10	21767	0

Total rows: 10    Query complete 00:00:47.802

## Q12(NQ14)

```
SELECT rev_actor, SUM(rev_minor_edit) AS total_minor_edits
FROM text JOIN revision ON old_id=rev_id
WHERE text_embedding {op} '{q}' < {d}
GROUP BY rev_actor
ORDER BY total_minor_edits DESC;
```

This is an **Aggregation on a Text Vector Range Search**.

- Filtering:** It finds *all* text segments in the `text` table that are semantically similar to the query vector within a specific distance threshold (`< {d}`). This is a pure range search.
- Joining & Aggregating:** It joins the matching texts to the `revision` table to identify the author (`rev_actor`) and sums up the number of minor edits they have made on these specific texts.

## Indexed Translation: Not Possible

This query uses a **distance threshold (`< {d}`)** rather than a count limit `k`.

- The `VECTOR_SEARCH` index function requires a `TOP_N` parameter. It cannot find "all rows within distance X".
- Therefore, we **cannot** use the vector index.
- We must use `VECTOR_DISTANCE` in the `WHERE` clause, which will force a full table scan.

## SQLServer

```
USE index_hybench_500k_new; -- Or HyBenchDB
GO
```

```
-- 1. Define parameters
```

```
DECLARE @distance_threshold FLOAT = 0.5; -- {d}
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding
FROM dbo.text
WHERE text_embedding IS NOT NULL
ORDER BY old_id;
```

```
PRINT '--- Running Q12 (NQ14?): Aggregation on Text Range Search (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT
    r.rev_user_text AS rev_actor,
    SUM(CAST(r.rev_minor_edit AS INT)) AS total_minor_edits
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id -- Join text to revision
WHERE
    -- Range Search Filter (Index cannot be used)
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) < @distance_threshold
GROUP BY
    r.rev_user_text
ORDER BY
    total_minor_edits DESC;
GO
```

## PostgreSQL

```
WITH q AS (
    SELECT text_embedding AS query_vec
```

```

FROM text
WHERE text_embedding IS NOT NULL
ORDER BY old_id
LIMIT 1
)
SELECT
    r.rev_user_text AS rev_actor,
    SUM(r.rev_minor_edit::int) AS total_minor_edits
FROM
    text t
JOIN
    revision r ON t.old_id = r.rev_id
JOIN
    q ON TRUE
WHERE
    (t.text_embedding <=> q.query_vec) < 0.5
GROUP BY
    r.rev_user_text
ORDER BY
    total_minor_edits DESC;

```

	rev_actor text	total_minor_edits bigint
1	8	111967
2	5	45992
3	38	9843
4	34	6145
5	20	5382
6	51	4667
7	31	3536
8	223	3453
9	233	3049

Total rows: 47607    Query complete 00:00:36.657

### Q13 (NQ9)

```

SELECT year, old_id, distance
FROM (

```

```

SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}'
AS distance,
    ROW_NUMBER() OVER (
        PARTITION BY EXTRACT (YEAR FROM rev_timestamp)
        ORDER BY text_embedding {op} '{q}', old_id
    ) AS rank FROM text JOIN revision ON old_id = rev_id
WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH}
) AS ranked_pages
WHERE rank <= {k}
ORDER BY year DESC,
distance ASC;

```

## Explanation of Q13 (NQ9)

This is a **Partitioned k-NN Search**.

1. **Filtering:** It selects revisions within a specific year range (**{YEARL}** to **{YEARH}**).
2. **Partitioning:** It groups the results by **year**.
3. **Ranking:** Inside *each* year group, it ranks the text segments by their similarity to the query vector.
4. **Selection:** It keeps only the top **{k}** most similar texts for **each year**. (e.g., "The top 10 matches from 2010, the top 10 from 2011, etc.").

## Indexed Translation: Not Possible

The **VECTOR\_SEARCH** index function finds the **global** top **k** matches across the entire dataset. It cannot find "top **k** per group" (partitioned search). To achieve this logic, we must calculate the distance for all relevant rows and then use a window function (**ROW\_NUMBER**) to sort them. This requires **VECTOR\_DISTANCE**, forcing a table scan.

SLServer

```

USE index_hybench_500k_new; -- Or HyBenchDB
GO

```

```

-- 1. Define parameters
DECLARE @k INT = 10; -- Top k per year
DECLARE @year_low INT = 2010;
DECLARE @year_high INT = 2015;
DECLARE @query_vector VECTOR(384);

```

```

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding
FROM dbo.text
WHERE text_embedding IS NOT NULL
ORDER BY old_id;

PRINT '--- Running NQ9: Partitioned k-NN (Top K per Year). ---';
PRINT '--- This will be SLOW (scan + sort). ---';

-- 3. Run the query
SELECT
    ranked_pages.year,
    ranked_pages.old_id,
    ranked_pages.distance
FROM (
    SELECT
        t.old_id,
        LEFT(r.rev_timestamp, 4) AS [year], -- Extract Year
        VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance,
        -- Partition by Year and Rank by Distance
        ROW_NUMBER() OVER (
            PARTITION BY LEFT(r.rev_timestamp, 4)
            ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC,
            t.old_id ASC
        ) AS rank
    FROM
        dbo.text AS t
    JOIN
        dbo.revision AS r ON t.old_id = r.rev_text_id
    WHERE
        -- Filter date range first to reduce rows for the window function
        CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
) AS ranked_pages
WHERE
    ranked_pages.rank <= @k -- Keep only Top K per year
ORDER BY
    ranked_pages.year DESC,
    ranked_pages.distance ASC;
GO

```

PostgreSQL

```
WITH q AS (
    -- 1. Get the query vector
    SELECT text_embedding AS query_vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
),
ranked AS (
    SELECT
        t.old_id,
        -- 2. Extract year safely
        SUBSTRING(r.rev_timestamp, 1, 4)::int AS year,
        -- 3. Compute cosine distance
        (t.text_embedding <=> q.query_vec) AS distance,
        -- 4. Partitioned ranking: top K per year
        ROW_NUMBER() OVER (
            PARTITION BY SUBSTRING(r.rev_timestamp, 1, 4)
            ORDER BY (t.text_embedding <=> q.query_vec), t.old_id
        ) AS rank
    FROM
        text t
    JOIN
        revision r ON t.old_id = r.rev_id
    JOIN
        q ON TRUE
    WHERE
        -- 5. Year range filter
        SUBSTRING(r.rev_timestamp, 1, 4)::int BETWEEN 2010 AND 2015
)
-- 6. Select only the top K per year
SELECT
    year,
    old_id,
    distance
FROM
    ranked
```

```

WHERE
rank <= 10
ORDER BY
year DESC,
distance ASC;

```

	year integer 	old_id [PK] integer	distance double precision 
1	2015	445250	0.03537188180083606
2	2015	369998	0.03544613293887311
3	2015	235439	0.037083590738070216
4	2015	417845	0.03710060248636404
5	2015	22176	0.03944911432444731
6	2015	93930	0.04030204129861881
7	2015	113784	0.04084489592680751
8	2015	407762	0.04406655637489265
9	2015	320756	0.044833324148659814
10	2015	313290	0.04494015673644014
Total rows: 60		Query complete 00:00:24.856	

## Q14 (NQ10)

```

SELECT year, old_id, distance
FROM (
  SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}'
  AS distance
  FROM text
  JOIN revision ON old_id = rev_id
  WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH} AND
text_embedding {op} '{q}'<= {d1}
) AS ranked_pages
ORDER BY year DESC,
distance ASC;

```

## Explanation of Q14 (NQ10)

This is a **Filtered Vector Range Search**.

1. **Filtering:** It selects text revisions based on a specific year range (`{YEARL}` to `{YEARH}`).
2. **Vector Threshold:** Unlike a k-NN search (which finds the "top 10"), this query finds **all** texts that are closer than a specific distance `{d1}` to the query vector.
3. **Sorting:** It orders the results by year and then by similarity distance.

## Indexed Translation: Not Possible

Because this query relies on a **distance threshold** (`<= {d1}`) rather than a fixed number of results (`LIMIT {k}`), we **cannot** use the `VECTOR_SEARCH` index function.

The index is designed to "find the closest X items," not "find everything within X distance." We must use `VECTOR_DISTANCE` in the `WHERE` clause, which forces a table scan.

SQLServer

```
USE index_hybench_500k_new; -- Or HyBenchDB  
GO
```

```
-- 1. Define parameters  
DECLARE @distance_threshold FLOAT = 0.5; -- {d1}  
DECLARE @year_low INT = 2010;  
DECLARE @year_high INT = 2015;  
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector  
SELECT TOP 1 @query_vector = text_embedding  
FROM dbo.text  
WHERE text_embedding IS NOT NULL  
ORDER BY old_id;
```

```
PRINT '--- Running NQ10: Filtered Range Search (Slow). ---';
```

```
-- 3. Run the query  
SELECT  
    LEFT(r.rev_timestamp, 4) AS [year], -- Extract Year  
    t.old_id,  
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance  
FROM  
    dbo.text AS t  
JOIN  
    dbo.revision AS r ON t.old_id = r.rev_text_id
```

```

WHERE
-- Filter 1: Date Range
CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
-- Filter 2: Vector Range (Index cannot be used)
AND VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) <=
@distance_threshold
ORDER BY
[year] DESC,
distance ASC;
GO

```

PostgreSQL

```

WITH q AS (
-- 1. Grab the query vector (same as SQL Server)
SELECT text_embedding AS query_vec
FROM text
WHERE text_embedding IS NOT NULL
ORDER BY old_id
LIMIT 1
)
SELECT
-- Extract year safely (works for any timestamp format)
SUBSTRING(r.rev_timestamp, 1, 4)::int AS year,
t.old_id,
-- Compute cosine distance
(t.text_embedding <=> q.query_vec) AS distance
FROM
text t
JOIN
revision r ON t.old_id = r.rev_id
JOIN
q ON TRUE
WHERE
-- Year range filter
SUBSTRING(r.rev_timestamp, 1, 4)::int BETWEEN 2010 AND 2015

-- Distance threshold filter (range search → no index)
AND (t.text_embedding <=> q.query_vec) <= 0.5
ORDER BY
year DESC,
distance ASC;

```

	<b>year</b> integer 	<b>old_id</b> [PK] integer 	<b>distance</b> double precision 
1	2015	445250	0.03537188180083606
2	2015	369998	0.03544613293887311
3	2015	235439	0.037083590738070216
4	2015	417845	0.03710060248636404
5	2015	22176	0.03944911432444731
6	2015	93930	0.04030204129861881
7	2015	113784	0.04084489592680751
8	2015	407762	0.04406655637489265
9	2015	320756	0.044833324148659814

Total rows: 54537    Query complete 00:00:17.539

## Q36 NQ16

```
SELECT old_id,old_text
FROM text
ORDER BY LEAST (
    text_embedding {op} '{q1}',
    text_embedding {op} '{q2}'
),old_id
LIMIT {k};
```

### Explanation of NQ16 (Query 36)

This is a **Multi-Target k-NN Search**.

- **Logic:** Instead of finding rows close to just *one* query vector, it finds rows that are close to *either* Query Vector A ({q1}) *or* Query Vector B ({q2}).
- **Ranking:** For every row, it calculates two distances (to q1 and q2), picks the *smaller* (LEAST) of the two, and uses that "best case" distance to rank the results.
- **Use Case:** "Find articles similar to 'Apples' OR 'Oranges'."

Indexed Translation: Possible 

We **can** use the vector index (VECTOR\_SEARCH) for this, but we have to be clever. VECTOR\_SEARCH only accepts *one* query vector at a time.

To solve this efficiently:

1. Run VECTOR\_SEARCH for {q1} to get its top {k} matches.
2. Run VECTOR\_SEARCH for {q2} to get *its* top {k} matches.
3. **Combine** the results (UNION), remove duplicates (group by ID), and pick the final top {k}.

This is much faster than scanning the whole table because we only look at the "best" candidates from both searches.

---

## NQ16 Translation (Indexed & Optimized)

This script uses the **Union Strategy** to leverage the index.

SQL

```
USE index_hybench_500k_new; -- Use your indexed DB
GO

-- 1. Enable Preview Features
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;
GO

-- 2. Define parameters
DECLARE @k INT = 10;
DECLARE @query_vector_1 VECTOR(384); -- {q1}
DECLARE @query_vector_2 VECTOR(384); -- {q2}

-- 3. Grab two real vectors to simulate {q1} and {q2}
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY old_id;
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY old_id DESC; --
Different vector

PRINT '--- Running NQ16: Multi-Target k-NN (Indexed Union). ---';

-- 4. Run the optimized query
SELECT TOP (@k)
    c.old_id,
    c.old_text,
    MIN(c.distance) AS best_distance -- Pick the better of the two scores
FROM
(
    -- Search 1: Nearest to Q1
```

```

SELECT
    t.old_id,
    t.old_text,
    s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.text AS t,
    COLUMN = text_embedding,
    SIMILAR_TO = @query_vector_1,
    METRIC = 'cosine',
    TOP_N = @k
) AS s

UNION ALL

-- Search 2: Nearest to Q2
SELECT
    t.old_id,
    t.old_text,
    s.distance
FROM VECTOR_SEARCH(
    TABLE = dbo.text AS t,
    COLUMN = text_embedding,
    SIMILAR_TO = @query_vector_2,
    METRIC = 'cosine',
    TOP_N = @k
) AS s
) AS c
GROUP BY
    c.old_id, c.old_text -- Deduplicate (in case a row is close to BOTH)
ORDER BY
    best_distance ASC,
    c.old_id ASC;
GO

```

PostgreSQL

```

WITH
-- 1. Fetch two different query vectors
q1 AS (
    SELECT text_embedding AS vec1
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1

```

```

),
q2 AS (
    SELECT text_embedding AS vec2
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id DESC
    LIMIT 1
),
-- 2. First k-NN search (restricted to old_id <= 500k)
top_q1 AS (
    SELECT
        t.old_id,
        t.old_text,
        cosine_distance(t.text_embedding, q1.vec1) AS distance
    FROM text t, q1
    WHERE t.old_id <= 500000
    ORDER BY t.text_embedding <-> q1.vec1
    LIMIT 10
),
-- 3. Second k-NN search (restricted to old_id <= 500k)
top_q2 AS (
    SELECT
        t.old_id,
        t.old_text,
        cosine_distance(t.text_embedding, q2.vec2) AS distance
    FROM text t, q2
    WHERE t.old_id <= 500000
    ORDER BY t.text_embedding <-> q2.vec2
    LIMIT 10
)
-- 4. Combine + deduplicate + take final top 10
SELECT
    old_id,
    old_text,
    MIN(distance) AS best_distance
FROM (
    SELECT * FROM top_q1
    UNION ALL
    SELECT * FROM top_q2
) AS combined
GROUP BY old_id, old_text

```

```
ORDER BY best_distance ASC, old_id ASC
LIMIT 10;
```

	old_id integer	old_text text	best_distance double precision
1	1	<strong>MediaWiki has been installed.<...>	0
2	5035	{{Short description Fictional comic book...}}	0.004352683654726186
3	464511	{{Short description Comic book character...}}	0.006089330839471052
4	17000	{{short description Fictional character a...}}	0.006352403449987998
5	50867	{{short description Superhero in Quality ...}}	0.00754830140638052
6	75081	{{about the comics character  Paula Bro...}}	0.007992976477836033
7	52875	{{Short description 1982 American anim...}}	0.008494332930275683
8	460417	{{short description Fictional character cr...}}	0.0085555192347605
9	426672	{{Infobox comics character\\n  image ...}}	0.008676697378588782
10	10613	{{Short description American comic arti...}}	0.009044614757879654

Total rows: 10    Query complete 00:00:38.409    CRLF    Ln 4, Col 31

### Q37 NQ18

```
SELECT old_id,old_text
FROM text
WHERE old_id not in (
    SELECT old_id
    FROM text
    ORDER BY text_embedding {op} '{q2}'
    LIMIT {k})
ORDER BY text_embedding {op} '{q1}'
LIMIT {k};
```

### Explanation of NQ18 (Query 37/38)

This is a **k-NN Search with an Exclusion List** (or "Near A but not Near B").

1. **Inner Query (Exclusion):** It finds the top  $\{k\}$  items closest to Query Vector 2 ( $\{q2\}$ ). These are the "forbidden" items.

2. **Outer Query (Search):** It finds the top  $\{k\}$  items closest to Query Vector 1 ( $\{q1\}$ ), *excluding* any items found in step 1.
3. **Use Case:** "Show me articles similar to 'Machine Learning' (Q1) but DO NOT show me articles similar to 'Deep Learning' (Q2)."

## Indexed Translation: Possible ✓

We **can** use the vector index (VECTOR\_SEARCH) for **both** parts of this query efficiently.

1. **Inner Part:** We use VECTOR\_SEARCH to quickly identify the ID list for  $\{q2\}$ .
2. **Outer Part:** We use VECTOR\_SEARCH to find candidates for  $\{q1\}$ .
3. **Filter:** We use a standard WHERE ... NOT IN clause to filter the results.

**Note:** Because VECTOR\_SEARCH is a post-filter, if the top 10 matches for Q1 contain items that are also in the top 10 for Q2, they will be removed, and you might get fewer than  $\{k\}$  results (e.g., 8 or 9). To guarantee exactly  $\{k\}$ , you would typically ask the outer index for  $k * 2$  items and then limit the final SELECT to  $k$ . I have implemented this "buffer" strategy below to ensure correctness.

## NQ18 Translation (Indexed)

SQL

```
USE index_hybench_100k; -- Use your indexed DB
GO

-- 1. Enable Preview Features
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;
GO

-- 2. Define parameters
DECLARE @k INT = 10;
DECLARE @k_buffer INT = 20; -- Fetch extra for Q1 to account for exclusions
DECLARE @query_vector_1 VECTOR(384); -- {q1}
DECLARE @query_vector_2 VECTOR(384); -- {q2}

-- 3. Grab two real vectors
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY old_id;
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY old_id DESC;

PRINT '--- Running NQ18: k-NN with Exclusion List (Indexed). ---';

-- 4. Run the query
SELECT TOP (@k) -- Final limit to ensure we return exactly k
    t.old_id,
    t.old_text
```

```

FROM
-- Outer Search: Find top matches for Q1 (with buffer)
VECTOR_SEARCH(
    TABLE = dbo.text AS t,
    COLUMN = text_embedding,
    SIMILAR_TO = @query_vector_1,
    METRIC = 'cosine',
    TOP_N = @k_buffer
) AS s1
WHERE
s1.old_id NOT IN (
    -- Inner Search: Find "Forbidden" IDs close to Q2
    SELECT s2.old_id
    FROM VECTOR_SEARCH(
        TABLE = dbo.text AS t2, -- Dummy alias for inner search
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector_2,
        METRIC = 'cosine',
        TOP_N = @k
    ) AS s2
)
ORDER BY
s1.distance ASC,
t.old_id ASC;
GO

```

PostgreSQL

```

WITH
-- Bring q1 and q2 vectors
q1 AS (
    SELECT text_embedding AS vec1
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
),
q2 AS (
    SELECT text_embedding AS vec2
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id DESC
    LIMIT 1
),

```

```
-- Inner (forbidden) set: top k nearest to q2
forbidden AS (
    SELECT
        t.old_id
    FROM
        text t, q2
    ORDER BY
        t.text_embedding <-> q2.vec2
    LIMIT 10
)

-- Outer query: top k nearest to q1 but excluding forbidden
SELECT
    t.old_id,
    t.old_text,
    (t.text_embedding <-> q1.vec1) AS distance
FROM
    text t, q1
WHERE
    t.old_id NOT IN (SELECT old_id FROM forbidden)
ORDER BY
    distance ASC,
    t.old_id ASC
LIMIT 10;
```

	old_id [PK] integer	old_text text		distance double precision	
1	1	<strong>MediaWiki has been installed.<...		0	
2	367891	Name: Marcus M. Edvall\\n\\nCompany:...		1.0504963428301113	
3	107909	Health forums refers to a [http://en.wiki...		1.1748077316435	
4	135968	I am "SaintHermit". \\nl don't wann...		1.1896324085238592	
5	201240	[https://www.deviantart.com/kazumikik...		1.2079093233016673	
6	80021	"<b> Hello! </b>"\\n\\nThank you for co...		1.2084256608263209	
7	315742	I am an entrepreneur and software devel...		1.2557662525921343	
8	337087	-- Noneluder.com ==\\n\\nLaunched on...		1.2687445447832661	
9	446771	Links solicited (and pending):\\n* [http:/...		1.2742917056923384	
10	186867	--Useful links==\\n[[tools:~dispenser/vi...		1.2742926411865054	

Total rows: 10    Query complete 00:00:28.390    CRLF    Ln 53, Col 1

Q38 NQ19

```
SELECT old_id,old_text
FROM text
WHERE text_embedding {op} '{q2}' > {d}
ORDER BY text_embedding {op} '{q1}'
LIMIT {k};
```

### Explanation of NQ19

This is a **k-NN Search with a "Dissimilarity" Filter**.

- **Logic:**
  1. **Search:** Find the top  $\{k\}$  text segments most similar to Query Vector 1 ( $\{q1\}$ ).
  2. **Filter:** *Exclude* any results that are too close to Query Vector 2 ( $\{q2\}$ ). Specifically, the distance to  $\{q2\}$  must be *greater than*  $\{d\}$ .
- **Use Case:** "Find articles about 'Apples' ( $\{q1\}$ ) but exclude anything related to 'Computers' ( $\{q2\}$ )."

### Indexed Translation: Not Possible

We **cannot** use the vector index (VECTOR\_SEARCH) for this query.

- **Reason:** The index is a **Post-filter**. If we ask VECTOR\_SEARCH for the top 10 matches for 'Apples', it will return the 10 closest items. If 5 of those happen to be about 'Computers' (and thus fail the  $\{q2\} > \{d\}$  check), we are left with only 5 results. The index cannot "look ahead" to find the next 5 valid matches to fill the quota.
- **The Fix:** We must scan the table, calculate both distances for every row, apply the filter, and then sort.

## NQ19 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @k INT = 10;
DECLARE @distance_threshold FLOAT = 0.5; -- {d} (Minimum distance from q2)
DECLARE @query_vector_1 VECTOR(384); -- {q1} (Target)
DECLARE @query_vector_2 VECTOR(384); -- {q2} (Avoid)

-- 2. Grab two real vectors
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY old_id;
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY old_id DESC;

PRINT '--- Running NQ19: k-NN with Dissimilarity Filter (Slow). ---';

-- 3. Run the query
SELECT TOP (@k)
    t.old_id,
    t.old_text
FROM
    dbo.text AS t
WHERE
    -- Filter: Must be "far" from q2
    VECTOR_DISTANCE('cosine', @query_vector_2, t.text_embedding) > @distance_threshold
ORDER BY
    -- Rank: Must be "close" to q1
    VECTOR_DISTANCE('cosine', @query_vector_1, t.text_embedding) ASC,
    t.old_id ASC;
GO
```

PostgreSQL

```
WITH
q1 AS (
    SELECT text_embedding AS v1
```

```
FROM text
WHERE text_embedding IS NOT NULL
ORDER BY old_id
LIMIT 1
),
q2 AS (
    SELECT text_embedding AS v2
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id DESC
    LIMIT 1
)

SELECT
    t.old_id,
    t.old_text
FROM text t, q1, q2
WHERE
    -- filter: must be far from q2
    cosine_distance(t.text_embedding, q2.v2) > 0.5  -- {d}
ORDER BY
    -- rank: close to q1
    cosine_distance(t.text_embedding, q1.v1) ASC,
    t.old_id ASC
LIMIT 10;  -- {k}
```

	old_id [PK] integer	old_text
1	179554	Sandbox editnotice test
2	107345	Hi welcome to my user page
3	76557	WEElcome welcome welcome to solarats page
4	441375	Add an article for Management Frame Protect...
5	402616	Page under construction
6	456369	Test page for my drafts.
7	449684	Parking lot for my user page
8	435886	New member, hello world!
9	341134	Welcome to my page.
10	420189	Welcome to my page.

Total rows: 10    Query complete 00:00:40.818    CRLF

### Q39 NQ17

```
SELECT old_id,old_text
FROM text
ORDER BY GREATEST (
    text_embedding {op} '{q1}',
    text_embedding {op} '{q2}'
),old_id
LIMIT {k};
```

### Explanation of NQ17

This is a **Minimax k-NN Search** (minimizing the maximum distance).

- **Logic:** For every row, it calculates the distance to  $\{q1\}$  and the distance to  $\{q2\}$ . It takes the **larger** (GREATEST) of the two values and sorts by that.
- **Goal:** It finds items that are reasonably close to *both* vectors. It penalizes items that are very close to one but very far from the other.
- **Use Case:** "Find articles that bridge the gap between 'History' and 'Science'."

**Indexed Translation: Not Possible**

We **cannot** use the vector index.

- **Reason:** The sorting logic (`GREATEST(d1, d2)`) depends on the exact distance to *both* query vectors for every single row.
  - A row might be the 500th closest to `{q1}` and the 500th closest to `{q2}`, making it the "winner" for this query.
  - However, `VECTOR_SEARCH` only finds the global "Top N" for a *single* vector. It would never find that "balanced" row if it wasn't in the top 10 for `{q1}` or `{q2}` individually.
- **The Fix:** We must calculate both distances for all rows and sort them manually (Full Table Scan).

## NQ17 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @k INT = 10;
DECLARE @query_vector_1 VECTOR(384); -- {q1}
DECLARE @query_vector_2 VECTOR(384); -- {q2}

-- 2. Grab two real vectors
SELECT TOP 1 @query_vector_1 = text_embedding FROM dbo.text ORDER BY old_id;
SELECT TOP 1 @query_vector_2 = text_embedding FROM dbo.text ORDER BY old_id DESC;

PRINT '--- Running NQ17: Minimax k-NN (Slow). ---';

-- 3. Run the query
SELECT TOP (@k)
    t.old_id,
    t.old_text
FROM
    dbo.text AS t
ORDER BY
    -- Sort by the GREATEST of the two distances (Minimax)
    GREATEST(
        VECTOR_DISTANCE('cosine', @query_vector_1, t.text_embedding),
        VECTOR_DISTANCE('cosine', @query_vector_2, t.text_embedding)
    ) ASC,
```

```
t.old_id ASC;  
GO
```

PostgreSQL

```
WITH  
q1 AS (  
    SELECT text_embedding AS v1  
    FROM text  
    WHERE text_embedding IS NOT NULL  
    ORDER BY old_id  
    LIMIT 1  
)  
,  
q2 AS (  
    SELECT text_embedding AS v2  
    FROM text  
    WHERE text_embedding IS NOT NULL  
    ORDER BY old_id DESC  
    LIMIT 1  
)  
  
SELECT  
    t.old_id,  
    t.old_text  
FROM text t, q1, q2  
ORDER BY  
    GREATEST(  
        cosine_distance(t.text_embedding, q1.v1),  
        cosine_distance(t.text_embedding, q2.v2)  
) ASC,  
    t.old_id ASC  
LIMIT 10;
```

	old_id [PK] integer	old_text
1	25055	{{Short description Saint Lucian musician}}\\n"Clement Springer" is
2	81223	{{WikiProject banner shell class=Stub}\\n{{WikiProject Brands importance=low status=active}}
3	44742	[[Image:SRL.svg thumb right Shift register lookup table.]]\\n\\nA "shift register lookup table" is a type of
4	60854	<noinclude>This quickbar may be useful to anyone who actively uses the {{SRL}} template. It is located in the
5	52379	{{WikiProject banner shell class=List 1=\\n{{WikiProject Lists class=box name=List list=}}\\n{{WikiProject Lists class=box name=List list=}}
6	344203	{{Wikipedia:Wikipedia Signpost/Templates/Signpost-article-header-1 class=box list=}}
7	158007	{{User:COIBot/Summary/LinkReports}}\\nReporting statistics of links to external sources
8	331128	{{WikiProject banner shell class=Start}}\\n{{WikiProject Mathematics status=active}}
9	234881	\\n== December 2008 ==\\n\\n[[Image:Information.png 25px]] Welcome to the Java Cryptography Extension (JCE) API documentation! This page contains information about the JCE API, including its history, features, and how to use it.
10	316285	{{refimprove date=March 2016}}\\n\\nThe "Java Cryptography Extension (JCE)" is a Java API for managing cryptographic keys and certificates. It provides a standard interface for performing various cryptographic operations, such as encryption, decryption, digital signing, and verification. The JCE API is part of the Java Platform, Standard Edition (Java SE) and is available in the Java Development Kit (JDK).

Total rows: 10

Query complete 00:00:16.899

# IQ - Postgres

## Q16 IQ1

```
SELECT old_id,old_text  
FROM text  
ORDER BY text_embedding {op} '{q}', old_id  
LIMIT {r-l+1} OFFSET {l-1};
```

### Explanation of IQ1 (Q16)

This is a **Paginated k-NN Search**.

It retrieves a specific "slice" of the nearest neighbors (e.g., "results 11 through 20").

- ORDER BY ...: Ranks all items by similarity.
- LIMIT ... OFFSET ...: Skips the first {l-1} results and returns the next {r-l+1} results.

### Indexed Translation: Possible

We **can** use the vector index, but with a small adjustment for pagination logic.

The VECTOR\_SEARCH index function does not have an internal OFFSET parameter. It always starts from rank 1. To implement pagination (e.g., "get items 11-20"), we must:

1. Ask the index for the **Top 20** items (TOP\_N = {r}).
2. Use standard SQL OFFSET ... FETCH NEXT to discard the first 10 and keep the next 10.

This remains efficient for typical pagination (browsing the first few pages of results).

### IQ1 Translation (Indexed)

SQL

```
USE index_hybench_100k; -- Use your 100k indexed database  
GO
```

```
-- 1. Enable Preview Features
```

```
ALTER DATABASE SCOPED CONFIGURATION SET PREVIEW_FEATURES = ON;  
GO
```

```
-- 2. Define parameters (Pagination Logic)
```

```
DECLARE @l INT = 11; -- Start Rank (e.g., 11)  
DECLARE @r INT = 20; -- End Rank (e.g., 20)
```

```

DECLARE @query_vector VECTOR(384);

-- Calculate arguments for SQL Server
DECLARE @limit INT = @r - @l + 1; -- {r-l+1} (How many to fetch)
DECLARE @offset INT = @l - 1; -- {l-1} (How many to skip)
DECLARE @fetch_total INT = @r; -- We must fetch *up to* the end rank from the index

-- 3. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running IQ1: Paginated k-NN (Indexed). ---';

-- 4. Run the query
SELECT
    t.old_id,
    t.old_text
FROM
    VECTOR_SEARCH(
        TABLE = dbo.text AS t,
        COLUMN = text_embedding,
        SIMILAR_TO = @query_vector,
        METRIC = 'cosine',
        -- The Index must retrieve the full set up to the highest rank needed
        TOP_N = @fetch_total
    ) AS s
ORDER BY
    s.distance ASC,
    t.old_id ASC
-- Standard SQL Pagination applies here, after the index retrieval
OFFSET @offset ROWS
FETCH NEXT @limit ROWS ONLY;
GO

```

**Note:** This approach is efficient for early pages. If you ask for "Results 10,000 to 10,010", the index still has to work hard to find the top 10,010 items first. This is standard behavior for vector databases.

PostgresSQL  
WITH  
q AS (

```
SELECT text_embedding AS vec
FROM text
WHERE text_embedding IS NOT NULL
ORDER BY old_id
LIMIT 1
),

top_r AS (
    SELECT
        t.old_id,
        t.old_text,
        cosine_distance(t.text_embedding, q.vec) AS distance
    FROM text t, q
    ORDER BY t.text_embedding <-> q.vec
    LIMIT 20 -- r = 20
)

SELECT
    old_id,
    old_text
FROM top_r
ORDER BY distance ASC, old_id ASC
OFFSET 10 -- l-1 = 11-1 = 10
LIMIT 10; -- r-l+1 = 10 results (11..20)
```

	old_id [PK] integer	old_text
1	186867	==Useful links==\\n[[tools:~dispenser/view/Checklinks Toolserver:]]
2	345860	ϕ <a target="_top" href="http://element.searchpluswin.com/?cr
3	23171	{{WikiProject banner shell class=Stub living=no listas=Herrera, Dan
4	87731	"W Hotel" is a under construction hotel in [[Dubai Festival City]] in [
5	188321	"Anago" (ἀνάγω), pronounced "an-ag'-o" is a word from the [[Ancier
6	263211	{{Primary sources date=April 2009}}\\n\\n"Third Party Internet Acc
7	415696	== Alkiama ==\\n\\n== About Me ==\\nWorld Traveler currently e
8	140788	This is a page to note some changes I made to the empathy page b
9	28262	==Wikimedia Pennsylvania==\\nHello there!\\n\\nl'm writing to info
10	369998	== Hobart area articles ==\\n\\nHello in advance of a possible pro

Total rows: 10    Query complete 00:00:28.135    CRLF    Ln 27, Col 1

## Q17 IQ2

Query:

```
SELECT old_id, old_text FROM text WHERE text_embedding {op} '{q}' BETWEEN {d} AND {d*} ...
```

### Explanation

This is a Bounded Range Search (or Annulus Search).

It asks for all text vectors that fall within a specific distance band (e.g., "distance is greater than 0.2 but less than 0.5").

### Indexed Translation: Not Possible

Just like Query 2 and Query 6, this is a range search.

- VECTOR\_SEARCH (the index tool) requires a TOP\_N limit ("Get me 10 items").
- It **cannot** handle "Get me items where distance is between X and Y".
- Therefore, we **cannot** use the index. We must use VECTOR\_DISTANCE in the WHERE clause, which forces a full table scan.

### IQ2 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Use your 100k DB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @dist_min FLOAT = 0.2; -- {d}
```

```
DECLARE @dist_max FLOAT = 0.5; -- {d*}
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```
PRINT '--- Running IQ2: Bounded Range Search (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT
```

```
t.old_id,
```

```
t.old_text,
```

```
VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
```

```
FROM
```

```
dbo.text AS t
```

```
WHERE
```

```
-- Range Filter (Index cannot be used)
```

```
VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @dist_min AND  
@dist_max
```

```
ORDER BY
```

```
    distance ASC,
```

```
    t.old_id ASC;
```

```
GO
```

PostgreSQL

```
WITH q AS (
    SELECT text_embedding AS v
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
)
```

```
SELECT
    t.old_id,
    t.old_text,
    cosine_distance(t.text_embedding, q.v) AS distance
FROM text t, q
WHERE
    cosine_distance(t.text_embedding, q.v)
    BETWEEN 0.2 AND 0.5 -- d and d* values
ORDER BY
    distance ASC,
    t.old_id ASC;
```

	old_id [PK] integer	old_text
1	171065	== Summary ==\\nRafael Santa Maria della Pace church in Rome'
2	241246	[[Category:Roads in Florida by county Flagler County]]\\n[[Catego
3	390329	{{WikiProject banner shell class=Stub}}\\n{{WikiProject Canada ns
4	171596	{{tasks\\n  requests=\\n  copyedit=\\n  wikify=\\n  merge=\\n
5	342657	#REDIRECT [[Orders of magnitude (length)#10 zeptometres]]\\n\\n
6	427541	{{WikiProject banner shell class=Start}}\\n{{WikiProject Sailing  im
7	250440	{{WikiProject banner shell class=B}}\\n{{WikiProject Chemicals im
8	107854	{{WikiProject banner shell class=Redirect}}\\n{{WikiProject Canada ns
-----		
Total rows: 228544		Query complete 00:00:16.350
CPU E In 20 Col 1		

Q18 IQ3:

### Explanation of IQ3

This is a **Paginated k-NN Search with a Pre-filter**.

1. **Filtering:** It restricts the search space to pages shorter than a specific length (page\_len < {len}).
2. **Ranking:** It ranks the remaining pages by semantic similarity to the query vector.
3. **Pagination:** It retrieves a specific "slice" of results (e.g., "ranks 11 to 20") from that filtered, ranked list.

### Indexed Translation: Not Possible

Just like with Q3, we **cannot** use the vector index (VECTOR\_SEARCH) for this query.

- **Reason:** VECTOR\_SEARCH is a **Post-filter**. It finds the global top N matches first. If you request "The top 10 short pages," but the global top 10 matches are all *long* pages, VECTOR\_SEARCH would return them, the WHERE clause would filter them all out, and you would get 0 results (incorrect recall).
- **Correctness:** To strictly satisfy the condition "Find top k *among* short pages," we must filter first and then sort. This requires using VECTOR\_DISTANCE in the ORDER BY clause, which forces a table scan.

## IQ3 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @l INT = 11; -- Start Rank {l}
DECLARE @r INT = 20; -- End Rank {r}
DECLARE @query_vector VECTOR(384);

-- Calculate SQL pagination arguments
DECLARE @limit INT = @r - @l + 1; -- {r-l+1} (How many to fetch)
DECLARE @offset INT = @l - 1; -- {l-1} (How many to skip)

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;

PRINT '--- Running IQ3: Paginated k-NN with Pre-Filter (Slow). ---';

-- 3. Run the query
SELECT
    p.page_id,
    p.page_title
FROM
    dbo.page AS p
WHERE
    -- Pre-filter (Must happen before sorting)
    p.page_len < @page_len_limit
ORDER BY
    -- Full table scan required to sort filtered results
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
    p.page_id ASC
-- Standard SQL Pagination
OFFSET @offset ROWS
FETCH NEXT @limit ROWS ONLY;
GO
```

postgresql
WITH q AS (

```

SELECT page_embedding AS vec
FROM page
WHERE page_embedding IS NOT NULL
ORDER BY page_id
LIMIT 1
)

SELECT
    p.page_id,
    p.page_title,
    cosine_distance(p.page_embedding, q.vec) AS distance
FROM page p, q
WHERE
    p.page_len < 1000      -- {len}
ORDER BY
    distance ASC,
    p.page_id ASC
OFFSET 10                 -- l = 11 → offset = 10
LIMIT 10;                  -- r = 20 → r-l+1 = 10

```

	page_id [PK] integer	page_title	distance double precision
1	313164	Version_book	0.034062620015791634
2	487384	Guide_pattern	0.034325871850203526
3	90450	Wave_format	0.034801712888602454
4	276827	Single_entry	0.034864372051946924
5	44990	Auto_link	0.035630267423788275
6	433241	Argument_for...	0.03568905602844441
7	110494	Start_Point	0.03620466762685326
8	150502	Top_Deck	0.036211033473411924
9	393118	Tale_type	0.03621789727665392
10	135306	Text_format	0.036255871902625136

Total rows: 10    Query complete 00:00:07.630    CRLF

Q19 IQ4

```
SELECT page_id,page_title
FROM page
WHERE page_len < {len} AND page_embedding {op} '{q}' BETWEEN {d} AND {d*}
ORDER BY page_embedding {op} '{q}',page_id;
```

## Explanation of IQ4

This is a **Filtered Bounded Range Search**.

1. **Relational Filter:** It restricts the search to pages with a length less than {len}.
2. **Vector Range Filter:** It selects pages where the semantic distance to the query vector falls within a specific band (between {d} and {d\*}).
3. **Sorting:** It orders the results by distance and then by page ID.

## Indexed Translation: Not Possible

Just like with **IQ2** and **Q4**, this is a **Range Search** (specifically a bounded one).

- The `VECTOR_SEARCH` index function strictly requires a `TOP_N` parameter (e.g., "Top 10").
- It **cannot** be configured to return "all items within a distance range."
- Therefore, we **cannot** use the vector index. We must use `VECTOR_DISTANCE` in the `WHERE` clause, which forces a full table scan.

## IQ4 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

```
-- 1. Define parameters
```

```
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @dist_min FLOAT = 0.2; -- {d}
DECLARE @dist_max FLOAT = 0.5; -- {d*}
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running IQ4: Filtered Bounded Range Search (Slow). ---';
```

```
-- 3. Run the query
```

```

SELECT
    p.page_id,
    p.page_title
FROM
    dbo.page AS p
WHERE
    -- Relational Filter
    p.page_len < @page_len_limit
    -- Vector Range Filter (Index cannot be used)
    AND VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @dist_min
    AND @dist_max
ORDER BY
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
    p.page_id ASC;
GO

```

```

psotgreSQL
WITH q AS (
    SELECT page_embedding AS vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
)
SELECT
    p.page_id,
    p.page_title,
    cosine_distance(p.page_embedding, q.vec) AS distance
FROM page p, q
WHERE
    -- relational filter
    p.page_len < 1000          -- {len}
    -- vector bounded range filter
    AND cosine_distance(p.page_embedding, q.vec)
        BETWEEN 0.2 AND 0.5 -- {d} AND {d*}
ORDER BY
    distance ASC,
    p.page_id ASC;

```

	<b>page_id</b> [PK] integer	<b>page_title</b> text	<b>distance</b> double precision
1	151056	Government_of_Slovakia_1994–1...	0.20000128168326536
2	411822	Chicagoan_(disambiguation)	0.20000140129436728
3	466948	Nam_Ji-Hyeon	0.20000151232573227
4	144996	Mazor_Bahaina	0.20000248594124415
5	303034	Sockpuppet_investigations/Tyler...	0.20000255596721028
6	168238	Lodewijknoll	0.2000041064752447
7	37806	Jimvaglia1970	0.2000049439436552
8	351003	Portal:Germany/Selected_picture...	0.20000502784944385
9	145119	Railway_stations_in_Canada_by_c...	0.2000054562236282
Total rows: 165474		Query complete 00:00:05.222	

## Q20 IQ5

```
SELECT rev_id
FROM text JOIN revision ON old_id=rev_id
WHERE rev_timestamp>='{DATE_LOW}' AND rev_timestamp<='{DATE_HIGH}'
ORDER BY text_embedding {op} '{q}', old_id
LIMIT {r-l+1} OFFSET {l-1};
```

### Explanation of IQ5

This is a **Paginated k-NN Search with a Join and Pre-filter**.

1. **Join:** It joins the `text` and `revision` tables.
2. **Pre-filter:** It restricts the search to revisions within a specific date range (`rev_timestamp` between `{DATE_LOW}` and `{DATE_HIGH}`).
3. **Ranking:** It ranks the matching text segments by their semantic similarity to the query vector.
4. **Pagination:** It retrieves a specific "slice" of results (e.g., ranks 11 to 20) from this filtered, ranked list.

### Indexed Translation: Not Possible

Just like with **Q5** and **IQ3**, we **cannot** use the vector index for this query.

- **Reason:** `VECTOR_SEARCH` acts as a **Post-filter**. It finds the global top N matches first. If the global top matches fall outside your date range, `VECTOR_SEARCH` would

discard them and return fewer (or zero) results, violating the requirement to find the top matches *within* that specific date range.

- **Correctness:** To correctly find the top matches *among* the revisions in that date range, we must filter first and then sort. This requires `VECTOR_DISTANCE` in the `ORDER BY` clause, forcing a full table scan.

## IQ5 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @l INT = 11; -- Start Rank  
DECLARE @r INT = 20; -- End Rank  
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z';  
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z';  
DECLARE @query_vector VECTOR(384);
```

```
-- Calculate SQL pagination arguments
```

```
DECLARE @limit INT = @r - @l + 1; -- How many to fetch  
DECLARE @offset INT = @l - 1; -- How many to skip
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```
PRINT '--- Running IQ5: Paginated k-NN with Date Filter (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT  
    r.rev_id  
FROM  
    dbo.text AS t  
JOIN  
    dbo.revision AS r ON t.old_id = r.rev_text_id  
WHERE  
    -- Pre-filter (Must happen before sorting)  
    r.rev_timestamp >= @date_low  
    AND r.rev_timestamp <= @date_high  
ORDER BY
```

```
-- Full scan required to sort filtered results
VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC,
t.old_id ASC
-- Standard SQL Pagination
OFFSET @offset ROWS
FETCH NEXT @limit ROWS ONLY;
GO
```

postgreSQL

```
WITH q AS (
    SELECT text_embedding AS vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
)
SELECT
    r.rev_id,
    cosine_distance(t.text_embedding, q.vec) AS distance
FROM text t
JOIN revision r ON t.old_id = r.rev_text_id
CROSS JOIN q      -- IMPORTANT: add q here
WHERE
    r.rev_timestamp >= '2010-01-01T00:00:00Z'
    AND r.rev_timestamp <= '2015-01-01T00:00:00Z'
ORDER BY
    distance ASC,
    t.old_id ASC
OFFSET 10
LIMIT 10;
```

	rev_id [PK] integer	distance double precision
1	336045	0.038364478066705066
2	121653	0.038717759324270795
3	491312	0.039629059574538106
4	478957	0.04029688345143856
5	490434	0.04035694990451555
6	197122	0.0407054290367711
7	374324	0.04099440156050438
8	452798	0.0412815516572157
9	283418	0.04139696938564086
10	458573	0.04150126167800505

Total rows: 10    Query complete 00:00:07.992

## Q21 | Q6

```
SELECT rev_id
FROM text JOIN revision ON old_id=rev_id
WHERE text_embedding {op} '{q}' BETWEEN {d} AND {d*} AND
rev_timestamp>='{DATE_LOW}' AND rev_timestamp<='{DATE_HIGH}'
ORDER BY text_embedding {op} '{q}', old_id;
```

### Explanation of IQ6

This is a **Filtered Bounded Range Search with Join**.

- Join:** It joins the `text` and `revision` tables.
- Relational Filter:** It restricts the search to revisions within a specific date range (`rev_timestamp`).
- Vector Range Filter:** It selects text segments where the semantic distance to the query vector falls within a specific band (between `{d}` and `{d*}`).
- Sorting:** It orders the results by similarity distance.

### Indexed Translation: Not Possible

Just like with **IQ2** and **IQ4**, this is a **Range Search** (specifically a bounded/annulus search).

- The `VECTOR_SEARCH` index function strictly requires a `TOP_N` parameter. It cannot return "all items within a specific distance band."
- Therefore, we **cannot** use the vector index. We must use the scalar `VECTOR_DISTANCE` function in the `WHERE` clause, forcing a full table scan.

## IQ6 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @dist_min FLOAT = 0.2;    -- {d}
DECLARE @dist_max FLOAT = 0.5;    -- {d*}
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z'; -- {DATE_LOW}
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z'; -- {DATE_HIGH}
DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running IQ6: Filtered Bounded Range Search (Slow). ---';

-- 3. Run the query
SELECT
    r.rev_id
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
    -- Relational Filter: Date Range
    r.rev_timestamp >= @date_low
    AND r.rev_timestamp <= @date_high
    -- Vector Range Filter (Index cannot be used)
    AND VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @dist_min
    AND @dist_max
ORDER BY
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC,
    t.old_id ASC;
GO
```

PostgreSQL

```

WITH q AS (
    SELECT text_embedding AS vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
)

SELECT
    r.rev_id,
    cosine_distance(t.text_embedding, q.vec) AS distance
FROM text t
JOIN revision r ON t.old_id = r.rev_text_id
CROSS JOIN q
WHERE
    -- Relational filter: date range
    r.rev_timestamp >= '2010-01-01T00:00:00Z'
    AND r.rev_timestamp <= '2015-01-01T00:00:00Z'
    -- Vector bounded range filter
    AND cosine_distance(t.text_embedding, q.vec)
        BETWEEN 0.2 AND 0.5
ORDER BY
    distance ASC,
    t.old_id ASC;

```

	<b>rev_id</b> [PK] integer	<b>distance</b> double precision
1	483209	0.200055401144567
2	217999	0.20006815823929958
3	39862	0.20010006692475557
4	437690	0.20011175019085803
5	426331	0.20012478187992755
6	341271	0.20012587004809856
7	446114	0.20016518720379284
8	489304	0.20020704321841865
9	489360	0.20020704321841865
Total rows: 28150		Query complete 00:00:11.316

Q22 IQ7:

```
SELECT rev_actor,COUNT(*) AS cou
FROM (
    SELECT page_id
    FROM page
    WHERE page_len < {len}
    ORDER BY page_embedding {op} '{q}', page_id
    LIMIT {r-l+1} OFFSET {l-1}
) AS new_page JOIN revision
ON page_id=rev_page
GROUP BY rev_actor
ORDER BY cou DESC;
```

IQ7

## Explanation of IQ7

This is an **Aggregation on a Paginated k-NN Search with a Pre-filter**.

### 1. Inner Query (Search):

- **Filter:** Selects only pages shorter than {len}.
- **Rank:** Orders them by semantic similarity to the query vector.
- **Paginate:** Retrieves a specific slice of results (ranks {l} to {r}).

### 2. Outer Query (Aggregation):

- Joins those specific pages to the revision table.
- Counts the contributions per author (rev\_actor).

## Indexed Translation: Not Possible

Just like **IQ3**, this query combines a **Pre-filter** (page\_len < {len}) with **Pagination**.

- **The Issue:** The `VECTOR_SEARCH` index function is a **Post-filter**. It finds the global top matches first. If the global top matches are all "long" pages, `VECTOR_SEARCH` would return them, the `WHERE` clause would discard them, and you would end up with an empty or incomplete page of results.
- **The Fix:** To guarantee you get a full page of "short" articles, we must filter first and then sort. This requires using `VECTOR_DISTANCE` in the `ORDER BY` clause, forcing a full table scan.

## IQ7 Translation (Non-Indexed)

## SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @l INT = 11; -- Start Rank {l}
DECLARE @r INT = 20; -- End Rank {r}
DECLARE @query_vector VECTOR(384);

-- Calculate SQL pagination arguments
DECLARE @limit INT = @r - @l + 1; -- {r-l+1} (How many to fetch)
DECLARE @offset INT = @l - 1; -- {l-1} (How many to skip)

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding FROM dbo.page ORDER BY page_id;

PRINT '--- Running IQ7: Aggregation on Paginated Pre-filtered Search (Slow). ---';

-- 3. Run the query
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM
    (
        -- Inner Query: Paginated k-NN with Pre-filter
        SELECT
            p.page_id
        FROM
            dbo.page AS p
        WHERE
            p.page_len < @page_len_limit -- Pre-filter
        ORDER BY
            VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
            p.page_id ASC
        OFFSET @offset ROWS
        FETCH NEXT @limit ROWS ONLY
    ) AS new_page
JOIN
```

```
dbo.revision AS r ON new_page.page_id = r.rev_page
GROUP BY
    r.rev_user_text
ORDER BY
    cou DESC;
GO
```

PostgreSQL

```
WITH q AS (
    SELECT page_embedding AS vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
)

SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM (
    -- Inner query: Pre-filter → Sort → Paginate
    SELECT
        p.page_id,
        cosine_distance(p.page_embedding, q.vec) AS distance
    FROM page p, q
    WHERE p.page_len < 1000           -- {len}
    ORDER BY distance ASC, p.page_id ASC
    OFFSET 10                         -- l = 11 → offset = 10
    LIMIT 10                          -- r = 20 → r-l+1 = 10
) AS new_page
JOIN revision r ON new_page.page_id = r.rev_page
GROUP BY r.rev_user_text
ORDER BY cou DESC;
```

	rev_actor	cou
	text	bigint
1	2379	3
2	3357	2
3	2459	1
4	14377	1
5	34	1
6	4742	1
7	8457	1

Q23 IQ8:

```
SELECT rev_actor,COUNT(*) AS cou
FROM page JOIN revision ON page_id=rev_page
WHERE page_len < {len} AND page_embedding {op} '{q}' BETWEEN {d} AND {d*}
GROUP BY rev_actor
ORDER BY cou DESC;
```

### Explanation of IQ8

This is an **Aggregation on a Filtered Bounded Range Search**.

1. **Filtering:** It selects pages that meet two criteria:
  - o **Relational:** Length is less than {len}.
  - o **Vector:** Semantic distance to the query vector is within a specific band (between {d} and {d\*}).
2. **Joining & Aggregating:** It joins matching pages to the revision table and counts the number of revisions per author (rev\_actor).

### Indexed Translation: Not Possible

This is a **Bounded Range Search** (finding all items within a distance range).

- The VECTOR\_SEARCH index function strictly requires a TOP\_N limit. It cannot return "all items between distance X and Y."
- Therefore, we **cannot** use the vector index. We must use VECTOR\_DISTANCE in the WHERE clause, which forces a full table scan.

### IQ8 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
```

```
GO
```

```
-- 1. Define parameters
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @dist_min FLOAT = 0.2; -- {d}
DECLARE @dist_max FLOAT = 0.5; -- {d*}
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running IQ8: Aggregation on Filtered Bounded Range Search (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM
    dbo.page AS p
JOIN
    dbo.revision AS r ON p.page_id = r.rev_page
WHERE
    -- Relational Filter
    p.page_len < @page_len_limit
    -- Vector Range Filter (Index cannot be used)
    AND VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @dist_min
    AND @dist_max
GROUP BY
    r.rev_user_text
ORDER BY
    cou DESC;
GO
```

postgresql

```
WITH q AS (
    SELECT page_embedding AS vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
)
```

```

SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM page p
JOIN revision r ON p.page_id = r.rev_page
CROSS JOIN q
WHERE
    -- Relational filter
    p.page_len < 1000          -- {len}
    -- Vector bounded range filter
    AND cosine_distance(p.page_embedding, q.vec)
        BETWEEN 0.2 AND 0.5    -- {d} AND {d*}
GROUP BY
    r.rev_user_text
ORDER BY
    cou DESC;

```

	rev_actor text	cou bigint
1	8	86939
2	5	9623
3	7	6078
4	233	3980
5	38	3640
6	532	2616
7	34	2193
8	20	1362
9	2379	1141

Total rows: 10015    Query complete 00:00:04.931

Q24 IQ9:

```

SELECT year, old_id, distance
FROM (
    SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}'
    AS distance, ROW_NUMBER() OVER (

```

```

        PARTITION BY EXTRACT (YEAR FROM rev_timestamp)
        ORDER BY text_embedding {op} '{q}', old_id
    ) AS rank FROM text JOIN revision ON old_id = rev_id
    WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH}
) AS ranked_pages
WHERE rank between {l} and {r}
ORDER BY year DESC,
distance ASC;

```

## Explanation of IQ9

This is a **Paginated Partitioned k-NN Search**.

- Filtering:** It selects revisions within a specific year range ({YEARL} to {YEARH}).
- Partitioning:** It groups the results by **year**.
- Ranking:** Inside *each* year group, it ranks the text segments by their similarity to the query vector.
- Pagination:** Unlike NQ9 (which took the "Top K"), this query takes a **specific slice** of ranks (e.g., "ranks 11 to 20") for *each* year.

## Indexed Translation: Not Possible

The `VECTOR_SEARCH` index function finds the **global** top `N` matches. It cannot handle:

- Partitioning:** Finding the top `N per category` (year).
- Offset/Slicing:** Finding *only* ranks `{l}` through `{r}` without retrieving the top ranks first.

To achieve "per-year" ranking, we must use a window function (`ROW_NUMBER()`) combined with `VECTOR_DISTANCE`, which requires calculating the distance for all matching rows (a table scan).

## IQ9 Translation (Non-Indexed)

SQL

```

USE index_hybench_100k; -- Or HyBenchDB
GO

```

-- 1. Define parameters

```

DECLARE @l INT = 11; -- Start Rank {l}
DECLARE @r INT = 20; -- End Rank {r}
DECLARE @year_low INT = 2010; -- {YEARL}
DECLARE @year_high INT = 2015; -- {YEARH}
DECLARE @query_vector VECTOR(384);

```

-- 2. Grab a real vector

```

SELECT TOP 1 @query_vector = text_embedding

```

```

FROM dbo.text
WHERE text_embedding IS NOT NULL
ORDER BY old_id;

PRINT '--- Running IQ9: Paginated Partitioned k-NN (Slow). ---';

-- 3. Run the query
SELECT
    ranked_pages.[year],
    ranked_pages.old_id,
    ranked_pages.distance
FROM (
    SELECT
        t.old_id,
        LEFT(r.rev_timestamp, 4) AS [year], -- Robust Year Extraction
        VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance,
        -- Partition by Year and Rank by Distance
        ROW_NUMBER() OVER (
            PARTITION BY LEFT(r.rev_timestamp, 4)
            ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC, t.old_id
        ) AS rank
    FROM
        dbo.text AS t
    JOIN
        dbo.revision AS r ON t.old_id = r.rev_text_id
    WHERE
        -- Filter date range first to reduce rows for the window function
        CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
) AS ranked_pages
WHERE
    ranked_pages.rank BETWEEN @l AND @r -- Select the specific slice
ORDER BY
    ranked_pages.[year] DESC,
    ranked_pages.distance ASC;
GO
PostgreSQL

WITH q AS (
    SELECT text_embedding AS vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
),

```

```
ranked_pages AS (
    SELECT
        t.old_id,
        EXTRACT(YEAR FROM r.rev_timestamp::timestamptz) AS year,
        cosine_distance(t.text_embedding, q.vec) AS distance,
        ROW_NUMBER() OVER (
            PARTITION BY EXTRACT(YEAR FROM r.rev_timestamp::timestamptz)
            ORDER BY
                cosine_distance(t.text_embedding, q.vec) ASC,
                t.old_id ASC
        ) AS rank
    FROM text t
    JOIN revision r ON t.old_id = r.rev_text_id
    CROSS JOIN q
    WHERE
        EXTRACT(YEAR FROM r.rev_timestamp::timestamptz)
        BETWEEN 2010 AND 2015
)
SELECT
    year,
    old_id,
    distance
FROM ranked_pages
WHERE rank BETWEEN 11 AND 20
ORDER BY year DESC, distance ASC;
```

	year numeric	old_id [PK] integer	distance double precision
1	2015	497111	0.04557626729642694
2	2015	211717	0.04642130498993324
3	2015	453686	0.04655050655118498
4	2015	164811	0.046804908961300185
5	2015	361003	0.04685678977969687
6	2015	431966	0.04688019235883134
7	2015	248667	0.04694162465444307
8	2015	373115	0.047107999425779035
9	2015	363371	0.047224799788576854
10	2015	332615	0.047418303342435686

Total rows: 60    Query complete 00:00:14.840    CRLF

Q25 IQ 10:

```

SELECT year, old_id, distance
FROM (
    SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding
    {op} '{q}' AS distance
    FROM text JOIN revision ON old_id = rev_id
    WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND
    {YEARH} AND text_embedding {op} '{q}' BETWEEN {d} AND {d*}
) AS ranked_pages
ORDER BY year DESC,
distance ASC;

```

## Explanation of IQ10

This is a **Filtered Bounded Range Search**.

1. **Filtering (Relational)**: It selects text revisions within a specific year range (`{YEARL}` to `{YEARH}`).

2. **Filtering (Vector):** It selects text segments where the semantic distance to the query vector falls within a specific band (between  $\{d\}$  and  $\{d^*\}$ ).
3. **Sorting:** It orders the results by year and then by similarity distance.

## Indexed Translation: Not Possible

This is a **Bounded Range Search** (finding all items within a distance band).

- The `VECTOR_SEARCH` index function strictly requires a `TOP_N` limit. It cannot return "all items between distance X and Y."
- Therefore, we **cannot** use the vector index. We must use `VECTOR_DISTANCE` in the `WHERE` clause, which forces a full table scan.

## IQ10 Translation (Non-Indexed)

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @dist_min FLOAT = 0.2;      -- {d}
DECLARE @dist_max FLOAT = 0.5;      -- {d*}
DECLARE @year_low INT = 2010;       -- {YEARL}
DECLARE @year_high INT = 2015;      -- {YEARH}
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = text_embedding
FROM dbo.text
WHERE text_embedding IS NOT NULL
ORDER BY old_id;
```

```
PRINT '--- Running IQ10: Filtered Bounded Range Search (Slow). ---';
```

-- 3. Run the query

```
SELECT
    LEFT(r.rev_timestamp, 4) AS [year], -- Robust Year Extraction
    t.old_id,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
```

```

-- Filter 1: Date Range
CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
-- Filter 2: Vector Range (Index cannot be used)
AND VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @dist_min
AND @dist_max
ORDER BY
[year] DESC,
distance ASC;
GO

```

PostgreSQL

```

WITH q AS (
    SELECT text_embedding AS vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
)
SELECT
    EXTRACT(YEAR FROM r.rev_timestamp::timestamptz) AS year,
    t.old_id,
    cosine_distance(t.text_embedding, q.vec) AS distance
FROM text t
JOIN revision r ON t.old_id = r.rev_text_id
CROSS JOIN q
WHERE
    -- Filter 1: year range
    EXTRACT(YEAR FROM r.rev_timestamp::timestamptz)
        BETWEEN 2010 AND 2015 -- {YEARL},{YEARH}
    -- Filter 2: distance range (bounded / annulus)
    AND cosine_distance(t.text_embedding, q.vec)
        BETWEEN 0.2 AND 0.5 -- {d},{d*}
ORDER BY
    year DESC,
    distance ASC;

```

	year numeric	old_id [PK] integer	distance double precision
1	2015	3410	0.2000989029693535
2	2015	294144	0.2001051946600888
3	2015	403807	0.2001558120783481
4	2015	45567	0.2004688535283239
5	2015	240099	0.20107197732405135
6	2015	118388	0.20157059364174723
7	2015	207411	0.20212097261536555
8	2015	91	0.202173313100251
9	2015	87531	0.20222029105279693

Total rows: 30500    Query complete 00:00:15.812

# SQ - PostgreSQL

## Q26 SQ1

```
SELECT *
FROM (
    SELECT old_id, old_text, ROW_NUMBER() OVER () AS rank
    FROM (
        SELECT old_id, old_text
        FROM text
        ORDER BY text_embedding {op} '{q}'
    ) AS ordered_limited
) AS ranked
WHERE rank IN ({r_1}, {r_2}, ..., {r_n});
```

### Explanation of SQ1

This is a **Specific Rank Selection Query**.

- **Logic:** It ranks *all* text segments by semantic similarity to the query vector and then retrieves only the items at specific positions (ranks), such as "the 1st, the 5th, and the 10th match."
- **Use Case:** This is often used for probing the quality of search results at different depths without fetching the entire list.

### Indexed Translation: Not Possible

We **cannot** effectively use the vector index (VECTOR\_SEARCH) here.

- **Reason:** The index is designed to return the **Top N** contiguous items (1, 2, 3... N). It cannot natively "skip" to pick specific ranks like 50 or 100 without fetching everything before them.
- **Current Status:** Since your index creation is blocked by the bug anyway, we must use the VECTOR\_DISTANCE function with a standard ROW\_NUMBER() window function. This forces a full table scan and sort.

### SQ1 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

```
-- 1. Define parameters
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector to search with
```

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```

PRINT '--- Running SQ1: Specific Rank Selection (Slow). ---';

-- 3. Run the query
SELECT
    ranked.old_id,
    ranked.old_text,
    ranked.rank
FROM (
    SELECT
        t.old_id,
        t.old_text,
        -- Calculate Rank based on Distance
        ROW_NUMBER() OVER (
            ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC, t.old_id
        ) AS rank
    FROM
        dbo.text AS t
) AS ranked
WHERE
    -- Filter for specific ranks {r_1, r_2, ...}
    -- Replace this list with your specific {r} values
    rank IN (1, 5, 10, 50, 100)
ORDER BY
    rank ASC;
GO

```

## PostgreSQL

```

WITH q AS (
    -- load one real query vector (SQL Server TOP 1)
    SELECT text_embedding AS vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
),
ranked AS (
    SELECT
        t.old_id,
        t.old_text,
        ROW_NUMBER() OVER (

```

```

        ORDER BY
            cosine_distance(t.text_embedding, q.vec) ASC,
            t.old_id ASC
    ) AS rank
FROM text t
CROSS JOIN q
)

```

```

SELECT
    old_id,
    old_text,
    rank
FROM ranked
WHERE rank IN (1, 5, 10, 50, 100) -- {r1, r2, ..., rn}
ORDER BY rank;

```

Showing rows: 1 to 5  Page No: 1 of 1  

	old_id [PK] integer 	old_text text 	rank bigint 
1	1	<strong>MediaWiki has been inst...	1
2	201240	[https://www.deviantart.com/kazu...	5
3	337087	-- Noneluder.com ==\n\nLaunc...	10
4	226024	"JNBridge" is a software vendor, f...	50
5	263379	\n== July 2009 ==\n\n[[Image:I...	100

Total rows: 5 | Query complete 00:00:28.355 | CRLF | L

Q27 SQ2

**Explanation of SQ2**

This is a **Multi-Range Search** (or Multi-Annulus Search).

- **Logic:** Instead of looking for items within a single distance band (like IQ2), this query looks for items that fall into *any* of several specific distance bands (e.g., "very close" OR "moderately far").
- **Use Case:** This might be used to sample data from different strata of similarity (e.g., finding examples of "exact matches," "near matches," and "far matches" in one go).

## Indexed Translation: Not Possible

As with all other **Range Searches** we have encountered (Q2, Q6, IQ2, etc.), we **cannot** use the vector index (VECTOR\_SEARCH) here.

- **Reason:** The index requires a strict TOP\_N limit. It does not support "distance thresholds," let alone *multiple* disjoint thresholds.
- **Impact:** We must use VECTOR\_DISTANCE in the WHERE clause with multiple OR conditions. This forces a full table scan.

## SQ2 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO

-- 1. Define parameters
-- Range 1
DECLARE @d1_min FLOAT = 0.1;
DECLARE @d1_max FLOAT = 0.2;
-- Range 2
DECLARE @d2_min FLOAT = 0.4;
DECLARE @d2_max FLOAT = 0.5;
-- Query Vector
DECLARE @query_vector VECTOR(384);

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;

PRINT '--- Running SQ2: Multi-Range Search (Slow). ---';

-- 3. Run the query
SELECT
    t.old_id,
    t.old_text,
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance
FROM
    dbo.text AS t
```

```

WHERE
-- Range 1
(VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d1_min AND
@d1_max)
OR
-- Range 2
(VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d2_min AND
@d2_max)
-- Add more OR conditions as needed
ORDER BY
distance ASC,
t.old_id ASC;
GO

```

PostgreSQL

```

WITH q AS (
SELECT text_embedding AS vec
FROM text
WHERE text_embedding IS NOT NULL
ORDER BY old_id
LIMIT 1
)

SELECT
t.old_id,
t.old_text,
cosine_distance(t.text_embedding, q.vec) AS distance
FROM text t
CROSS JOIN q
WHERE
-- Range 1
cosine_distance(t.text_embedding, q.vec) BETWEEN 0.1 AND 0.2
OR
-- Range 2
cosine_distance(t.text_embedding, q.vec) BETWEEN 0.4 AND 0.5
ORDER BY
distance ASC,
t.old_id ASC;

```

	old_id [PK] integer	old_text text	distance double precision
1	235168	{{testcases notice}}\\n\\n== Live ...	0.10000096693857541
2	102429	{{Userbox Who am I? align=lef...}}	0.1000010860138687
3	261181	{{Talk header}}\\n{{WikiProject ba...}}	0.10000109069814889
4	488526	{{Short description 1971 film by H...}}	0.10000119626380921
5	394709	{{Navbox\\n name = Footer Olymp...}}	0.10000121041102883
6	136690	{{Administrators' noticeboard nav...}}	0.10000198265236415
7	487036	== Visa Black Card ==\\nAfter I ad...	0.10000206063320893
8	205088	"Copied from the image's main pa..."	0.10000261514835596
9	116441	{{Short description Australian acti...}}	0.10000286191479069
Total rows: 102212		Query complete 00:00:25.513	

CRI F In 24 Col 1

## Q28 SQ3

```

SELECT *
FROM (
    SELECT page_id, page_title, ROW_NUMBER() OVER () AS rank
    FROM (
        SELECT page_id, page_title
        FROM page
        WHERE page_len < {len}
        ORDER BY page_embedding {op} '{q}', page_id
    ) AS top_k
) AS ranked
WHERE rank IN ({r_1}, {r_2}, ..., {r_n});

```

### Explanation of SQ3

This is a **Specific Rank Selection with Pre-filter**.

- **Logic:**
  1. **Filter:** Selects only pages shorter than {len}.
  2. **Rank:** Orders the remaining pages by semantic similarity to the query vector.
  3. **Select:** Retrieves only specific positions from this ranked list (e.g., "Get the 1st, 10th, and 50th matching short page").
- **Use Case:** Checking result quality at specific depths for a filtered subset of data.

## Indexed Translation: Not Possible

Just like **Q3** and **IQ3**, this query has a **Pre-filter** (`page_len < {len}`).

- **The Issue:** The `VECTOR_SEARCH` index function is a **Post-filter**. It finds the global top matches first. If the global top matches are all "long" pages, they get filtered out later, destabilizing the ranking. You can't ask the index for "The 10th best *short* page."
- **The Fix:** We must scan the table, filter by length, calculate distances for the survivors, and then rank them.

## SQ3 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector to search with

```
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running SQ3: Specific Rank Selection with Pre-Filter (Slow). ---';
```

-- 3. Run the query

```
SELECT
    ranked.page_id,
    ranked.page_title,
    ranked.rank
FROM (
    SELECT
        p.page_id,
        p.page_title,
        -- Calculate Rank based on Distance
        ROW_NUMBER() OVER (
            ORDER BY VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
            p.page_id ASC
        ) AS rank
    FROM
        dbo.page AS p
    WHERE
        -- Pre-filter: Only consider short pages
```

```

    p.page_len < @page_len_limit
) AS ranked
WHERE
    -- Filter for specific ranks {r_1, r_2, ...}
    rank IN (1, 5, 10, 50, 100)
ORDER BY
    rank ASC;
GO
PostgreSQL
WITH q AS (
    -- Load a real query vector (equivalent to SQL Server TOP 1)
    SELECT page_embedding AS vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
),
ranked AS (
    SELECT
        p.page_id,
        p.page_title,
        ROW_NUMBER() OVER (
            ORDER BY
                cosine_distance(p.page_embedding, q.vec) ASC,
                p.page_id ASC
        ) AS rank
    FROM page p
    CROSS JOIN q
    WHERE p.page_len < 1000    -- {len}
)
SELECT
    page_id,
    page_title,
    rank
FROM ranked
WHERE rank IN (1, 5, 10, 50, 100)  -- {r1, r2, ...}
ORDER BY rank ASC;

```

	page_id [PK] integer	page_title text	rank bigint
1	1	Main_Page	1
2	458436	Content_format	5
3	122511	File_system	10
4	23276	Split_level	50
5	461389	Bath_Profile	100

Total rows: 5    Query complete 00:00:04.152    CRLF    L

## Q29 SQ4

```
SELECT page_id, page_title,
FROM page
WHERE page_len < {len} AND ((page_embedding {op} '{q}' BETWEEN {d_1} AND {d_1*})
OR (page_embedding {op} '{q}' BETWEEN {d_2} AND {d_2*})
OR .. OR (page_embedding {op} '{q}' BETWEEN {d_n} AND {d_n*}))
ORDER BY page_embedding {op} '{q}', page_id;
```

### Explanation of SQ4

This is a **Filtered Multi-Range Search**.

- **Logic:**
  1. **Relational Filter:** Selects only pages shorter than {len}.
  2. **Vector Multi-Range Filter:** Selects pages where the semantic distance falls into *any* of several specific bands (e.g., "very close" OR "somewhat far").
  3. **Sorting:** Orders results by distance.
- **Use Case:** Complex sampling or categorizing items into specific similarity buckets while excluding long documents.

### Indexed Translation: Not Possible

We **cannot** use the vector index (VECTOR\_SEARCH) for this query.

- **Reason 1 (Range Search):** The index requires a TOP\_N limit. It cannot handle "distance between X and Y".
- **Reason 2 (Pre-filter):** The index cannot apply the page\_len < {len} filter *before* searching the graph.
- **Reason 3 (Complex Logic):** The index cannot natively handle the OR logic between multiple distance ranges.

We must use VECTOR\_DISTANCE in the WHERE clause, forcing a full table scan.

## SQ4 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @query_vector VECTOR(384);
```

-- Range 1 ({d\_1} to {d\_1\*})

```
DECLARE @d1_min FLOAT = 0.1;
DECLARE @d1_max FLOAT = 0.2;
```

-- Range 2 ({d\_2} to {d\_2\*})

```
DECLARE @d2_min FLOAT = 0.4;
DECLARE @d2_max FLOAT = 0.5;
```

-- 2. Grab a real vector to search with

```
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;
```

```
PRINT '--- Running SQ4: Filtered Multi-Range Search (Slow). ---';
```

-- 3. Run the query

```
SELECT
    p.page_id,
    p.page_title,
    VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) AS distance
FROM
    dbo.page AS p
WHERE
    -- Filter 1: Relational (Pre-filter)
```

```

p.page_len < @page_len_limit

AND (
    -- Filter 2: Vector Range 1
    (VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @d1_min
AND @d1_max)
    OR
    -- Filter 3: Vector Range 2
    (VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @d2_min
AND @d2_max)
)
ORDER BY
    distance ASC,
    p.page_id ASC;
GO

```

PostgreSQL

```

WITH q AS (
    -- Load a real query vector (equivalent to SQL Server TOP 1)
    SELECT page_embedding AS vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
)

```

```

SELECT
    p.page_id,
    p.page_title,
    cosine_distance(p.page_embedding, q.vec) AS distance
FROM page p
CROSS JOIN q
WHERE
    -- Filter 1: Relational pre-filter
    p.page_len < 1000          -- {len}

```

```

AND (
    -- Vector Range 1
    cosine_distance(p.page_embedding, q.vec)
    BETWEEN 0.1 AND 0.2      -- {d1_min}, {d1_max}
)

```

OR

```

-- Vector Range 2
cosine_distance(p.page_embedding, q.vec)

```

```

        BETWEEN 0.4 AND 0.5    -- {d2_min}, {d2_max}

-- Add more ranges here:
-- OR cosine_distance(...) BETWEEN x AND y
)
ORDER BY
    distance ASC,
    p.page_id ASC;

```

	page_id [PK] integer	page_title text	distance double precision
1	173229	Venus'_hair	0.10000001079281795
2	418226	My_Hats_Collection	0.10000044726775437
3	446960	Southend_Seattle	0.10000084989378522
4	62797	Volt42	0.10000115081387628
5	479199	Priscilla_Pig	0.1000013379224719
6	479314	Priscilla_Pig	0.1000013379224719
7	335036	Umpire_abuse	0.10000282402954841
8	479881	Asia-related_lists	0.10000313930765359
9	443327	Paige_Brooks	0.10000663673490229
Total rows: 153876		Query complete 00:00:04.811	

### Q30 SQ5

```

SELECT rev_id
FROM (
    SELECT rev_id, ROW_NUMBER() OVER () AS rank
    FROM (
        SELECT rev_id
        FROM text JOIN revision ON old_id = rev_id
        WHERE rev_timestamp >= '{DATE_LOW}' AND rev_timestamp <= '{DATE_HIGH}'
        ORDER BY text_embedding {op} '{q}', old_id
    ) AS top_k
) AS ranked
WHERE rank IN ({r_1}, {r_2}, .., {r_n});

```

### Explanation of SQ5

This is a **Specific Rank Selection on a Filtered Join**.

1. **Filtering (Relational)**: It selects revisions within a specific date range (`{DATE_LOW}` to `{DATE_HIGH}`).
2. **Ranking**: It ranks the matching revisions by the semantic similarity of their text content to the query vector.
3. **Select Specific Ranks**: It returns only specific positions from this ranked list (e.g., "the 1st, 10th, and 100th matching revision in that date range").

## Indexed Translation: Not Possible

We **cannot** use the vector index (`VECTOR_SEARCH`) for this query.

- **Reason 1 (Pre-filter)**: The index is a Post-filter. It finds the global top matches first. If the global top matches fall outside the date range, `VECTOR_SEARCH` discards them, breaking the ranking logic for the specific date range.
- **Reason 2 (Rank Selection)**: The index cannot "jump" to specific ranks (e.g., "Get rank 50") without retrieving ranks 1-49 first.

We must use `VECTOR_DISTANCE` in the `ORDER BY` clause inside a window function, which forces a scan of all rows matching the date filter.

## SQ5 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z'; -- {DATE_LOW}
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z'; -- {DATE_HIGH}
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```
PRINT '--- Running SQ5: Specific Rank Selection with Date Filter (Slow). ---';
```

-- 3. Run the query

```
SELECT
    ranked.rev_id,
    ranked.rank
FROM (
    SELECT
        r.rev_id,
        -- Calculate Rank based on Distance
```

```

    ROW_NUMBER() OVER (
        ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC, t.old_id
ASC
    ) AS rank
FROM
    dbo.text AS t
JOIN
    dbo.revision AS r ON t.old_id = r.rev_text_id
WHERE
    -- Pre-filter: Date Range
    r.rev_timestamp >= @date_low
    AND r.rev_timestamp <= @date_high
) AS ranked
WHERE
    -- Filter for specific ranks {r_1, r_2, ...}
    rank IN (1, 5, 10, 50, 100)
ORDER BY
    rank ASC;
GO

```

```

postgresql
WITH q AS (
    -- Load a real query vector (same as SQL Server TOP 1)
    SELECT text_embedding AS vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
),
ranked AS (
    SELECT
        r.rev_id,
        ROW_NUMBER() OVER (
            ORDER BY
                cosine_distance(t.text_embedding, q.vec) ASC,
                t.old_id ASC
        ) AS rank
    FROM text t
    JOIN revision r
        ON t.old_id = r.rev_text_id
    CROSS JOIN q
    WHERE
        -- Date pre-filter (rev_timestamp is TEXT → cast to timestamptz)
        r.rev_timestamp::timestamptz >= '2010-01-01T00:00:00Z'
        AND r.rev_timestamp::timestamptz <= '2015-01-01T00:00:00Z'
)

```

```
)
SELECT
  rev_id,
  rank
FROM ranked
WHERE rank IN (1, 5, 10, 50, 100)  -- {r1, r2, ...}
ORDER BY rank ASC;
```

	rev_id [PK] integer	rank bigint
1	216418	1
2	415696	5
3	336045	10
4	316635	50
5	411430	100

Total rows: 5    Query complete 00:00:14.757

Q31 SQ6:

```

SELECT rev_id
FROM text JOIN revision ON old_id = rev_id
WHERE rev_timestamp >= '{DATE_LOW}' AND rev_timestamp <= '{DATE_HIGH}'
AND ((text_embedding {op} '{q}' BETWEEN {d_1} AND {d_1*})
OR (text_embedding {op} '{q}' BETWEEN {d_2} AND {d_2*}))
OR .. OR (text_embedding {op} '{q}' BETWEEN {d_n} AND {d_n*}))
ORDER BY text_embedding {op} '{q}', old_id;
USE index_hybench_100k; -- Or HyBenchDB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @date_low NVARCHAR(50) = '2010-01-01T00:00:00Z'; -- {DATE_LOW}  
DECLARE @date_high NVARCHAR(50) = '2015-01-01T00:00:00Z'; -- {DATE_HIGH}  
DECLARE @query_vector VECTOR(384);
```

```
-- Range 1 ({d_1} to {d_1*})
```

```
DECLARE @d1_min FLOAT = 0.1;  
DECLARE @d1_max FLOAT = 0.2;
```

```
-- Range 2 ({d_2} to {d_2*})
```

```
DECLARE @d2_min FLOAT = 0.4;  
DECLARE @d2_max FLOAT = 0.5;
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding FROM dbo.text ORDER BY old_id;
```

```
PRINT '--- Running SQ6: Filtered Multi-Range Search with Join (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT  
    r.rev_id  
FROM  
    dbo.text AS t  
JOIN  
    dbo.revision AS r ON t.old_id = r.rev_text_id  
WHERE  
    -- Filter 1: Relational Date Range  
    r.rev_timestamp >= @date_low  
    AND r.rev_timestamp <= @date_high  
  
    AND (  
        -- Filter 2: Vector Range 1  
        (VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d1_min  
        AND @d1_max)  
        OR  
        -- Filter 3: Vector Range 2  
        (VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) BETWEEN @d2_min  
        AND @d2_max)  
    )  
ORDER BY  
    VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC,  
    t.old_id ASC;
```

GO

```
PostgreSQL
WITH q AS (
    -- 1. Load query vector
    SELECT text_embedding AS vec
    FROM text
    WHERE text_embedding IS NOT NULL
    ORDER BY old_id
    LIMIT 1
)
SELECT
    r.rev_id,
    cosine_distance(t.text_embedding, q.vec) AS distance
FROM text t
JOIN revision r
    ON t.old_id = r.rev_text_id
CROSS JOIN q
WHERE
    -- 2. Date filter (must be before distance checks)
    r.rev_timestamp::timestamptz >= '2010-01-01T00:00:00Z'
    AND r.rev_timestamp::timestamptz <= '2015-01-01T00:00:00Z'
    -- 3. Multi-range cosine distance filtering
    AND (
        cosine_distance(t.text_embedding, q.vec) BETWEEN 0.1 AND 0.2
        OR
        cosine_distance(t.text_embedding, q.vec) BETWEEN 0.4 AND 0.5
        -- Add more OR ranges as needed
    )
ORDER BY
    distance ASC,
    t.old_id ASC;
```

	rev_id [PK] integer	distance double precision
1	235168	0.10000096693857541
2	487350	0.10001320007388903
3	242722	0.10002439840737765
4	25450	0.10004398098271772
5	490190	0.10005342451550481
6	384727	0.10005663414194255
7	439968	0.10006481864374472
8	465354	0.10007509909251078
9	387867	0.1000776669030371
10	489007	0.10008929504530717

Total rows: 9022    Query complete 00:00:15.423

Q32 SQ7:

```

SELECT rev_actor, COUNT(*) AS cou
FROM (
  SELECT page_id
  FROM (
    SELECT page_id, ROW_NUMBER() OVER () AS rank
    FROM (
      SELECT page_id
      FROM page
      WHERE page_len < {len}
      ORDER BY page_embedding {op} '{q}', page_id
    ) AS top_k
  ) AS ranked
  WHERE rank IN ({r_1}, {r_2}, ..., {r_n})
) AS filtered_page JOIN revision ON page_id = rev_page
GROUP BY rev_actor
ORDER BY cou DESC;

```

### Explanation of SQ7

This is an Aggregation on Specific Rank Selection with a Pre-filter.

1. **Filter & Rank (Inner):**
  - **Pre-filter:** It selects only pages shorter than {len}.
  - **Rank:** It orders these pages by semantic similarity to the query vector.
  - **Select Ranks:** It picks specific positions from this filtered list (e.g., "the 1st, 10th, and 50th best matching *short* page").
2. **Join & Aggregate (Outer):** It joins those specific pages to the revision table to count the contributions of each author (rev\_actor).

## Indexed Translation: Not Possible

We **cannot** use the vector index (VECTOR\_SEARCH) for this query.

- **Reason:** The index is a **Post-filter**. It finds the global top N matches first. If you ask for "Rank 1," the index finds the single closest page in the database. If that page happens to be *long* ( $> \{len\}$ ), it gets filtered out, and you are left with **Rank 1: NULL**. The index cannot "skip" the long pages during its search to find the "Rank 1 Short Page."
- **The Fix:** We must scan the table, apply the length filter first, and then sort the remaining rows to calculate the correct ranks.

## SQ7 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
GO
```

-- 1. Define parameters

```
DECLARE @page_len_limit INT = 1000; -- {len}
DECLARE @query_vector VECTOR(384);
```

-- 2. Grab a real vector

```
SELECT TOP 1 @query_vector = page_embedding FROM dbo.page ORDER BY page_id;
```

```
PRINT '--- Running SQ7: Aggregation on Specific Ranks with Pre-Filter (Slow). ---';
```

-- 3. Run the query

```
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM
(
    SELECT
        ranked.page_id
    FROM (
        SELECT
            p.page_id,
```

```

-- Calculate Rank based on Distance, considering only short pages
ROW_NUMBER() OVER (
    ORDER BY VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) ASC,
p.page_id ASC
) AS rank
FROM
dbo.page AS p
WHERE
p.page_len < @page_len_limit -- Pre-filter
) AS ranked
WHERE
-- Select specific ranks {r_1, r_2...}
ranked.rank IN (1, 5, 10, 50, 100)
) AS filtered_page
JOIN
dbo.revision AS r ON filtered_page.page_id = r.rev_page
GROUP BY
r.rev_user_text
ORDER BY
cou DESC;
GO

```

```

postgreSQL
WITH q AS (
    -- Load the query vector
    SELECT page_embedding AS vec
    FROM page
    WHERE page_embedding IS NOT NULL
    ORDER BY page_id
    LIMIT 1
),
ranked_pages AS (
    SELECT
        p.page_id,
        ROW_NUMBER() OVER (
            ORDER BY cosine_distance(p.page_embedding, q.vec), p.page_id
        ) AS rank
    FROM page p
    CROSS JOIN q
    WHERE
        p.page_len < 1000 -- {len}
),
filtered_page AS (
    SELECT page_id
    FROM ranked_pages

```

```
WHERE rank IN (1, 5, 10, 50, 100) -- {r1, r2, ...}
)
```

```
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM filtered_page fp
JOIN revision r
    ON fp.page_id = r.rev_page
GROUP BY r.rev_user_text
ORDER BY cou DESC;
```

	rev_actor text	cou bigint
1	18578	1
2	2	1
3	200	1
4	5	1
5	7	1

Total rows: 5    Query complete 00:00:04.658

Q33 SQ8:

```
SELECT rev_actor, COUNT(*) AS cou
FROM (
    SELECT page_id
    FROM page
    WHERE page_len < {len} AND
```

```

((page_embedding {op} '{q}' BETWEEN {d_1} AND {d_1*})
OR (page_embedding {op} '{q}' BETWEEN {d_2} AND {d_2*})
OR .. OR (page_embedding {op} '{q}' BETWEEN {d_n} AND {d_n*}))
) AS filtered_page
JOIN revision ON page_id = rev_page
GROUP BY rev_actor
ORDER BY cou DESC;

```

## Explanation of SQ8

This is an **Aggregation on a Filtered Multi-Range Search**.

### 1. Inner Logic (Search):

- **Relational Filter:** Selects only pages shorter than {len}.
- **Vector Multi-Range Filter:** Selects pages where the semantic distance to the query vector falls into *any* of several specific bands (e.g., "Distance is 0.1-0.2 OR 0.4-0.5").

### 2. Outer Logic (Aggregation):

- Joins the matching pages to the revision table.
- Counts the number of revisions made by each author (rev\_actor) on those specific pages.

## Indexed Translation: Not Possible

We **cannot** use the vector index (VECTOR\_SEARCH) for this query.

- **Reason 1 (Range Search):** The index requires a TOP\_N limit. It cannot find "all items within distance ranges."
- **Reason 2 (Complex Logic):** The index cannot natively handle the OR logic between multiple disjoint distance ranges.
- **Reason 3 (Pre-filter):** The index cannot apply the page\_len < {len} filter *before* traversing the graph.

We must use VECTOR\_DISTANCE in the WHERE clause with complex boolean logic, forcing a full table scan.

## SQ8 Translation (Non-Indexed)

SQL

```

USE index_hybench_100k; -- Or HyBenchDB
GO

```

-- 1. Define parameters

```
DECLARE @page_len_limit INT = 1000; -- {len}
```

```

DECLARE @query_vector VECTOR(384);

-- Range 1
DECLARE @d1_min FLOAT = 0.1;
DECLARE @d1_max FLOAT = 0.2;

-- Range 2
DECLARE @d2_min FLOAT = 0.4;
DECLARE @d2_max FLOAT = 0.5;

-- 2. Grab a real vector
SELECT TOP 1 @query_vector = page_embedding
FROM dbo.page
WHERE page_embedding IS NOT NULL
ORDER BY page_id;

PRINT '--- Running SQ8: Aggregation on Filtered Multi-Range Search (Slow). ---';

-- 3. Run the query
SELECT
    r.rev_user_text AS rev_actor,
    COUNT(*) AS cou
FROM
    dbo.page AS p
JOIN
    dbo.revision AS r ON p.page_id = r.rev_page
WHERE
    -- Relational Filter
    p.page_len < @page_len_limit

    AND (
        -- Multi-Range Vector Logic
        (VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @d1_min
        AND @d1_max)
        OR
        (VECTOR_DISTANCE('cosine', @query_vector, p.page_embedding) BETWEEN @d2_min
        AND @d2_max)
    )
GROUP BY

```

```
r.rev_user_text  
ORDER BY  
    cou DESC;  
GO
```

PostgreSQL

```
WITH q AS (  
    -- Load a query vector  
    SELECT page_embedding AS vec  
    FROM page  
    WHERE page_embedding IS NOT NULL  
    ORDER BY page_id  
    LIMIT 1  
)  
  
SELECT  
    r.rev_user_text AS rev_actor,  
    COUNT(*) AS cou  
FROM  
    page p  
JOIN  
    revision r ON p.page_id = r.rev_page  
CROSS JOIN  
    q  
WHERE  
    -- Relational filter  
    p.page_len < 1000  
  
    AND (  
        -- Vector Distance Range 1  
        cosine_distance(p.page_embedding, q.vec) BETWEEN 0.1 AND 0.2  
  
        OR  
  
        -- Vector Distance Range 2  
        cosine_distance(p.page_embedding, q.vec) BETWEEN 0.4 AND 0.5  
  
        -- Add more OR conditions for more ranges  
    )  
GROUP BY  
    r.rev_user_text
```

```
ORDER BY  
cou DESC;
```

	rev_actor	cou
1	5	20784
2	7	13907
3	38	5579
4	2379	4703
5	34	3530
6	20	2182
7	223	2048
8	31	1936
9	53	1677

Total rows: 25141    Query complete 00:00:06.224

Q34 SQ9:

```
SELECT year, old_id, distance  
  
FROM (  
  
    SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}' AS  
distance, ROW_NUMBER() OVER (  
  
        PARTITION BY EXTRACT (YEAR FROM rev_timestamp)  
  
        ORDER BY text_embedding {op} '{q}', old_id  
  
    ) AS rank FROM text JOIN revision ON old_id = rev_id  
  
    WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH}  
  
) AS ranked_pages  
  
WHERE rank in ({r_1},{r_2},..,{r_n})
```

```
ORDER BY year DESC, distance ASC;
```

## Explanation of SQ9

This is a **Partitioned Specific Rank Selection**.

- **Logic:**
  1. **Filter:** Selects revisions within a specific year range (`{YEARL}` to `{YEARH}`).
  2. **Partition:** Groups the results by `year`.
  3. **Rank:** Within each year, ranks the text segments by their semantic similarity to the query vector.
  4. **Select:** Picks only specific ranks from each year (e.g., "Get the 1st, 5th, and 10th best match for 2010, for 2011, etc.").
- **Use Case:** Sampling search quality at specific depths across different time periods.

## Indexed Translation: Not Possible

We **cannot** use the vector index (`VECTOR_SEARCH`) for this query.

- **Reason 1 (Partitioning):** The index finds the **global** top `N` matches. It cannot find "Top N per Year."
- **Reason 2 (Specific Ranks):** The index returns a contiguous block of results (1 to N). It cannot "skip" to fetch only specific ranks (like 5 and 10) without fetching 1-4 and 6-9.

We must use `VECTOR_DISTANCE` inside a window function (`ROW_NUMBER()`), which requires calculating the distance for all rows matching the date filter (a table scan).

## SQ9 Translation (Non-Indexed)

SQL

```
USE index_hybench_100k; -- Or HyBenchDB
```

```
GO
```

```
-- 1. Define parameters
```

```
DECLARE @year_low INT = 2010; -- {YEARL}
```

```
DECLARE @year_high INT = 2015; -- {YEARH}
```

```
DECLARE @query_vector VECTOR(384);
```

```
-- 2. Grab a real vector
```

```
SELECT TOP 1 @query_vector = text_embedding  
FROM dbo.text  
WHERE text_embedding IS NOT NULL  
ORDER BY old_id;
```

```
PRINT '--- Running SQ9: Partitioned Specific Rank Selection (Slow). ---';
```

```
-- 3. Run the query
```

```
SELECT  
    ranked_pages.[year],  
    ranked_pages.old_id,  
    ranked_pages.distance  
FROM (  
    SELECT  
        t.old_id,  
        LEFT(r.rev_timestamp, 4) AS [year], -- Robust Year Extraction  
        VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) AS distance,  
  
        -- Partition by Year and Rank by Distance  
        ROW_NUMBER() OVER (  
            PARTITION BY LEFT(r.rev_timestamp, 4)  
            ORDER BY VECTOR_DISTANCE('cosine', @query_vector, t.text_embedding) ASC, t.old_id  
            ASC  
        ) AS rank  
    FROM  
        dbo.text AS t
```

JOIN

```
dbo.revision AS r ON t.old_id = r.rev_text_id
```

WHERE

```
-- Filter date range first
```

```
CAST(LEFT(r.rev_timestamp, 4) AS INT) BETWEEN @year_low AND @year_high
```

```
) AS ranked_pages
```

WHERE

```
-- Filter for specific ranks {r_1, r_2...}
```

```
ranked_pages.rank IN (1, 5, 10, 20)
```

ORDER BY

```
ranked_pages.[year] DESC,
```

```
ranked_pages.distance ASC;
```

```
GO
```

postgreSQL

WITH q AS (

```
-- Load query vector
```

```
SELECT text_embedding AS vec
```

```
FROM text
```

```
WHERE text_embedding IS NOT NULL
```

```
ORDER BY old_id
```

```
LIMIT 1
```

```
),
```

ranked AS (

```
SELECT
```

```
t.old_id,
```

```
(LEFT(r.rev_timestamp, 4))::INT AS year,
```

```
cosine_distance(t.text_embedding, q.vec) AS distance,
```

```
ROW_NUMBER() OVER (
```

```
PARTITION BY (LEFT(r.rev_timestamp, 4))::INT
```

```
ORDER BY cosine_distance(t.text_embedding, q.vec), t.old_id
```

```
) AS rank
```

```

FROM text t
JOIN revision r
  ON t.old_id = r.rev_text_id
CROSS JOIN q
WHERE (LEFT(r.rev_timestamp, 4))::INT BETWEEN 2010 AND 2015 -- {YEARL}, {YEARH}
)

SELECT
  year,
  old_id,
  distance
FROM ranked
WHERE rank IN (1, 5, 10, 20) -- {r1, r2, r3...}
ORDER BY year DESC, distance ASC;

```

	year integer 	old_id [PK] integer 	distance double precision 
1	2015	445250	0.03537188180083606
2	2015	22176	0.03944911432444731
3	2015	313290	0.04494015673644014
4	2015	332615	0.047418303342435686
5	2014	415696	0.034354896615113484
6	2014	419619	0.0429145102970403
7	2014	400288	0.04334128939241744
8	2014	372727	0.04459825163350717
9	2013	446771	0.032752297660286867
10	2013	482810	0.04704013055108536
Total rows: 24		Query complete 00:00:14.819	

Q35 SQ 10:

```

SELECT year, old_id, distance
FROM (

```

```
SELECT old_id, EXTRACT (YEAR FROM rev_timestamp) AS year, text_embedding {op} '{q}'  
AS distance,  
    FROM text JOIN revision ON old_id = rev_id  
    WHERE EXTRACT (YEAR FROM rev_timestamp) BETWEEN {YEARL} AND {YEARH}  
        AND ((text_embedding {op} '{q}' BETWEEN {d_1} AND {d_1*})  
            OR (text_embedding {op} '{q}' BETWEEN {d_2} AND {d_2*})  
            OR .. OR (text_embedding {op} '{q}' BETWEEN {d_n} AND {d_n*}))  
) AS ranked_pages  
ORDER BY year DESC, distance ASC;
```

recall postgres

## Q1(NQ1)

```
SET ivfflat.probes = 50;

DO $$

DECLARE
    k_values INT[] := ARRAY[10, 20, 50, 100, 200, 500];
    max_k    INT;
    qvec     VECTOR(384);

    gt_start TIMESTAMP;
    gt_end   TIMESTAMP;
    full_ms  FLOAT;

    idx_start TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms   FLOAT;

    overlap INT;
    recall  FLOAT;
    accel   FLOAT;

    k INT;
BEGIN
-----
-- 1. Pick a NORMALIZED query vector
-----
SELECT l2_normalize(text_embedding_norm)
  INTO qvec
  FROM text
  ORDER BY old_id
  LIMIT 1;

SELECT MAX(x) FROM unnest(k_values) t(x)
  INTO max_k;

DROP TABLE IF EXISTS q1_results;
CREATE TEMP TABLE q1_results(
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
```

);

---

-- 2. Ground Truth (exact full-scan)

---

```
DROP TABLE IF EXISTS gt_q1;
CREATE TEMP TABLE gt_q1(id INT PRIMARY KEY, rank INT);
```

```
PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);
```

```
gt_start := clock_timestamp();
```

```
INSERT INTO gt_q1(id, rank)
SELECT old_id,
       ROW_NUMBER() OVER (
           ORDER BY text_embedding_norm <=> qvec, old_id)
FROM text
ORDER BY text_embedding_norm <=> qvec, old_id
LIMIT max_k;
```

```
gt_end := clock_timestamp();
```

```
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;
```

---

-- 3. Enable ANN index

---

```
PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);
```

---

-- 4. MAIN LOOP OVER ALL K VALUES

---

```
FOREACH k IN ARRAY k_values LOOP
```

---

-- Indexed timing

---

```
idx_start := clock_timestamp();
```

```
DROP TABLE IF EXISTS idx_q1;
CREATE TEMP TABLE idx_q1(id INT);
```

```
INSERT INTO idx_q1(id)
```

```

SELECT old_id
FROM text
ORDER BY text_embedding_norm <=> qvec, old_id
LIMIT k;

idx_end := clock_timestamp();
idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

-----
-- Recall@k
-----

SELECT COUNT(*) INTO overlap
FROM idx_q1 i
JOIN gt_q1 g USING(id)
WHERE g.rank <= k;

recall := overlap::FLOAT / k * 100.0;
accel := full_ms / idx_ms;

INSERT INTO q1_results VALUES (k, recall, idx_ms, accel);
END LOOP;

END $$;

SELECT * FROM q1_results ORDER BY k;

```

	k integer 	recall double precision 	index_time_ms double precision 	speedup double precision 
1	10	100	35.176	33.69342733682056
2	20	100	35.976	32.94418501223038
3	50	96	33.135	35.76882450580957
4	100	98	33.938	34.9225057457717
5	200	98.5	38.079	31.124766931904723
6	500	97.6	45.366	26.125292068950316

## Grid search

```
DROP TABLE IF EXISTS q1_grid_search_results;

CREATE TABLE q1_grid_search_results(
    lists INT,
    probes INT,
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
);

DO $$

DECLARE
    lists_values  INT[] := ARRAY[200, 400, 600, 800, 1000];
    probes_values INT[] := ARRAY[5, 10, 20, 50, 100];
    k_values      INT[] := ARRAY[10, 20, 50, 100, 200, 500];

    qvec          VECTOR(384);
    max_k         INT;

    lists INT;
    probes INT;
    k INT;

    gt_start  TIMESTAMP;
    gt_end    TIMESTAMP;
    full_scan_ms FLOAT;

    idx_start  TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms    FLOAT;

    overlap INT;
    recall   FLOAT;
    accel    FLOAT;

BEGIN
-----
-- 1. Pick query vector (already normalized column)
-----
SELECT text_embedding_norm INTO qvec
FROM text
```

```

ORDER BY old_id
LIMIT 1;

SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;

-----
-- 2. Ground Truth once
-----

DROP TABLE IF EXISTS gt_q1;
CREATE TEMP TABLE gt_q1(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_q1(id, rank)
SELECT old_id,
       ROW_NUMBER() OVER (
           ORDER BY (text_embedding_norm <=> qvec), old_id)
FROM text
ORDER BY (text_embedding_norm <=> qvec), old_id
LIMIT max_k;

gt_end := clock_timestamp();
full_scan_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;

-----
-- 3. GRID SEARCH
-----

FOREACH lists IN ARRAY lists_values LOOP
    RAISE NOTICE 'Rebuilding index with lists=%', lists;

    EXECUTE 'DROP INDEX IF EXISTS text_embedding_norm_ivf_idx';
    EXECUTE format(
        'CREATE INDEX text_embedding_norm_ivf_idx
        ON text USING ivfflat (text_embedding_norm vector_cosine_ops)
        WITH (lists = %s)',
        lists
    );
    EXECUTE 'ANALYZE text';

    FOREACH probes IN ARRAY probes_values LOOP

```

```

RAISE NOTICE 'Testing lists=%, probes=%', lists, probes;

EXECUTE format('SET ivfflat.probes = %s', probes);

PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);

FOREACH k IN ARRAY k_values LOOP
-----
-- Timed ANN search
-----
idx_start := clock_timestamp();

PERFORM 1 FROM (
    SELECT old_id
    FROM text
    ORDER BY text_embedding_norm <=> qvec, old_id
    LIMIT k
) sub;

idx_end := clock_timestamp();
idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

-----
-- Capture ANN IDs
-----
DROP TABLE IF EXISTS idx_q1;
CREATE TEMP TABLE idx_q1(id INT);

INSERT INTO idx_q1(id)
SELECT old_id
FROM text
ORDER BY text_embedding_norm <=> qvec, old_id
LIMIT k;

-----
-- Recall@k
-----
SELECT COUNT(*) INTO overlap
FROM idx_q1 i
JOIN gt_q1 g ON g.id = i.id
WHERE g.rank <= k;

recall := (overlap::FLOAT / k) * 100.0;

```

```

accel := full_scan_ms / idx_ms;

INSERT INTO q1_grid_search_results
VALUES (lists, probes, k, recall, idx_ms, accel);
END LOOP; -- k
END LOOP; -- probes
END LOOP; -- lists
END $$;

```

```

-- See best configs
SELECT *
FROM q1_grid_search_results
ORDER BY k, recall DESC, index_time_ms ASC;

```

### **lists, probes, k, recall, idx\_ms, accel**

800	20	10	100	9.222	400.80166991975716
800	50	10	100	18.455	200.2813871579518
600	50	10	100	26.063	141.81763419406823
1000	50	10	100	31.203	118.45633432682756
800	100	10	100	40.756	90.69076945725783
1000	100	10	100	41.702	88.6334708167474
600	100	10	100	45.633	80.99824688273837
400	50	10	100	53.97	68.48606633314805
400	100	10	100	70.067	52.75226568855525
200	20	10	100	85.976	42.9909858565181
200	50	10	100	158.997	23.246935476770002
200	100	10	100	216.281	17.08977210203393
1000	20	10	90	6.9	535.6801449275363
600	20	10	90	11.569	319.4911401158268
400	20	10	90	16.936	218.24474492205954
200	10	10	90	25.615	144.29798945930122
400	10	10	80	9.354	395.1457130639299
1000	10	10	70	5.385	686.3868152274838
600	10	10	70	8.297	445.48547667831747
800	10	10	70	8.928	414.00011200716847
200	5	10	70	22.328	165.54071121461843
1000	5	10	60	9.29	397.867922497309
400	5	10	60	11.843	312.09938360212783
600	5	10	50	15.468	238.95739591414534
800	5	10	10	12.062	306.43284695738686
1000	50	20	100	18.693	197.73139677954313
600	50	20	100	23.381	158.08532569180105
1000	100	20	100	32.88	112.41462895377128

600	100	20	100	56.357	65.58533988679312
200	50	20	100	84.977	43.49639314167363
200	100	20	100	118.728	31.131603328616674
800	100	20	95	37.866	97.61244916283738
400	50	20	95	49.677	74.40451315498119
400	100	20	95	65.593	56.35041848977787
200	20	20	95	113.758	32.49171926370014
1000	20	20	90	6.386	578.7962730974006
800	20	20	90	6.986	529.0857429144003
600	20	20	90	11.238	328.90131696031324
800	50	20	90	16.626	222.31402622398653
400	20	20	85	14.881	248.38337477320073
400	10	20	80	7.195	513.7168867268937
200	10	20	80	25.545	144.69340379722058
1000	10	20	75	4.615	800.9085590465872
600	10	20	75	6.121	603.8544355497468
1000	5	20	65	4.896	754.9413807189543
200	5	20	65	27.019	136.79977053184797
800	10	20	60	3.693	1000.8646087191985
400	5	20	60	7.877	469.2386695442428
600	5	20	50	6.221	594.1477254460698
800	5	20	20	5.319	694.9037413047566
1000	100	50	100	40.728	90.75311824788844
600	100	50	100	48.241	76.61932795754649
200	50	50	100	90.17	40.99138294332927
200	100	50	100	144.307	25.613400597337623
600	50	50	98	24.18	152.86157981803143
400	50	50	98	44.638	82.80373224606838
400	100	50	98	71.578	51.638673894213305
1000	50	50	96	17.437	211.97413545908125
800	100	50	96	48.496	76.2164508413065
200	20	50	96	82.574	44.76218906677647
800	50	50	92	19.989	184.91135124318376
800	20	50	86	8.821	419.02199297131847
600	20	50	86	11.811	312.9449665565998
1000	20	50	84	10.131	364.83989734478337
400	20	50	84	18.205	203.03174951936285
400	10	50	72	12.18	303.46412151067324
200	10	50	72	25.582	144.4841294660308
600	10	50	68	10.922	338.41723127632304
800	10	50	62	7.761	476.2521582270326
1000	10	50	62	9.821	376.35607371958054
200	5	50	62	51.467	71.81675636815824
400	5	50	52	16.71	221.19646918013166

1000	5	50	46	14.57	253.68517501715854
600	5	50	44	13.913	265.664702077194
800	5	50	26	55.066	67.12296153706461
1000	100	100	100	38.391	96.27759110208123
600	100	100	100	43.338	85.28757672250681
200	50	100	100	79.694	46.37981529349763
200	100	100	100	118.246	31.25850345889079
400	50	100	99	36.859	100.27925337095418
400	100	100	99	64.989	56.8741325455077
600	50	100	98	21.679	170.49647123944834
1000	50	100	98	24.23	152.546141147338
800	100	100	98	38.827	95.19646122543591
200	20	100	97	85.72	43.11937704153057
800	50	100	95	17.196	214.9449290532682
400	20	100	90	17.086	216.32874868313243
600	20	100	89	14.628	252.67931364506427
800	20	100	87	9.783	377.8179495042421
200	10	100	81	22.946	161.08223655539092
1000	20	100	76	10.69	345.76173994387284
400	10	100	76	21.003	175.98404989763367
600	10	100	69	8.858	417.2717317678934
200	5	100	67	67.744	54.561186230514885
800	10	100	65	9.44	391.54586864406787
1000	10	100	51	22.122	167.0822258385318
600	5	100	50	25.817	143.16895843823838
400	5	100	45	23.763	155.54403905230822
800	5	100	37	21.86	169.0847666971638
1000	5	100	33	26.235	140.88785972936918
1000	100	200	100	28.035	131.84209024433744
600	100	200	100	42.831	86.29714459153416
200	50	200	100	110.739	33.37751830881623
200	100	200	100	126.203	29.28767937370744
400	100	200	99.5	63.495	58.212347428931416
800	100	200	99	35.395	104.4269812120356
400	50	200	99	35.446	104.276730801783
1000	50	200	98.5	22.026	167.8104512848452
600	50	200	98.5	23.211	159.2431605704192
800	50	200	97	17.037	216.95093032810942
200	20	200	97	75.158	49.17896963729743
400	20	200	93	19.798	186.69527224972222
600	20	200	91.5	18.419	200.67283783050112
800	20	200	89	11.201	329.987768949201
200	10	200	86	28.981	127.53849073530934
1000	20	200	83	15.353	240.74728066175993

400	10	200	77.5	25.439	145.29631667911477
200	5	200	74.5	105.77	34.945570577668526
800	10	200	71.5	13.602	271.73893545066903
600	10	200	70	16.66	221.86032412965187
1000	10	200	57.99999999999999	42.421	87.1312085995144
600	5	200	53.5	52.791	70.01558977856075
400	5	200	50.5	45.367	81.47316331253997
800	5	200	45.5	38.713	95.47679074212797
1000	5	200	27	55.83	66.20442414472507
600	100	500	100	43.701	84.57914006544473
200	100	500	100	110.585	33.42399963828729
200	50	500	100	117	31.591393162393164
400	100	500	99.6	68.888	53.65510684008826
800	100	500	99.4	40.01	92.3817295676081
400	50	500	99.2	47.022	78.60561013993451
600	50	500	99	26.691	138.48087370274627
1000	100	500	99	35.039	105.48797054710465
800	50	500	97.6	19.892	185.81304041825862
1000	50	500	97.6	30.105	122.77671483142336
200	20	500	97.6	63.606	58.1107599911958
600	20	500	94.39999999999999	32.093	115.17131461689465
400	20	500	93.8	30.866	119.74965981986652
800	20	500	92	22.812	162.0284499386288
200	10	500	90.4	47.071	78.52378322109155
1000	20	500	86.8	43.938	84.1229232099777
200	5	500	81.39999999999999	207.555	17.808258052082582
800	10	500	79.60000000000001	31.264	118.22521110542478
400	10	500	79	36.846	100.31463388156111
600	10	500	72.6	32.117	115.08525080175609
1000	10	500	63.4	107.207	34.477161006277576
600	5	500	58.4	151.622	24.377682658189443
400	5	500	58.19999999999996	117.679	31.409112925840635
800	5	500	56.8	123.745	29.869433108408423
1000	5	500	19.8	121.823	30.3406827938895

## Q3(NQ3)

SET ivfflat.probes = 50;

DO \$\$

```

DECLARE
    k_values  INT[] := ARRAY[10, 20, 50, 100, 200];
    max_k     INT;

    qvec      VECTOR(384);
    len_filter INT := 1000;

    gt_start  TIMESTAMP;
    gt_end    TIMESTAMP;
    full_ms   FLOAT;

    idx_start TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms    FLOAT;

    overlap INT;
    recall  FLOAT;
    k INT;

BEGIN
    SELECT l2_normalize(page_embedding_norm)
    INTO qvec
    FROM page
    ORDER BY page_id
    LIMIT 1;

    SELECT MAX(x) FROM unnest(k_values) t(x)
    INTO max_k;

    DROP TABLE IF EXISTS q3_results;
    CREATE TEMP TABLE q3_results(
        k INT,
        recall FLOAT,
        index_time_ms FLOAT,
        speedup FLOAT
    );

    DROP TABLE IF EXISTS gt;
    CREATE TEMP TABLE gt(id INT PRIMARY KEY, rank INT);

    PERFORM set_config('enable_seqscan','on',true);
    PERFORM set_config('enable_indexscan','off',true);

    gt_start := clock_timestamp();

```

```

INSERT INTO gt(id, rank)
SELECT page_id,
    ROW_NUMBER() OVER (
        ORDER BY page_embedding_norm <=> qvec, page_id
    )
FROM page
WHERE page_len < len_filter
ORDER BY page_embedding_norm <=> qvec, page_id
LIMIT max_k;

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;

PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);

FOREACH k IN ARRAY k_values LOOP

    idx_start := clock_timestamp();

    DROP TABLE IF EXISTS idx;
    CREATE TEMP TABLE idx(id INT);

    INSERT INTO idx(id)
    SELECT page_id
    FROM page
    WHERE page_len < len_filter
    ORDER BY page_embedding_norm <=> qvec, page_id
    LIMIT k;

    idx_end := clock_timestamp();
    idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

    SELECT COUNT(*) INTO overlap
    FROM idx i
    JOIN gt g USING(id)
    WHERE g.rank <= k;

    recall := overlap::FLOAT / k * 100.0;

    INSERT INTO q3_results VALUES (
        k, recall, idx_ms, full_ms / idx_ms
    );
END LOOP;

```

```
END $$;
```

```
SELECT * FROM q3_results ORDER BY k;
```

	k integer	recall double precision	index_time_ms double precision	speedup double precision
1	10	100	24.539	670.369208199193
2	20	100	28.487	577.4630533225682
3	50	100	19.095	861.4920136161298
4	100	100	23.055	713.5194101062676
5	200	100	22.397	734.481850247801

## Q9(NQ11)

```
SET ivfflat.probes = 10;

DO $$
DECLARE
    k_values  INT[] := ARRAY[10, 20, 50, 100, 200];
    max_k      INT;
    qvec      VECTOR(384);
    gt_start  TIMESTAMP;
    gt_end    TIMESTAMP;
    full_ms   FLOAT;
    idx_start TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms    FLOAT;
    overlap   INT;
    recall    FLOAT;
    accel    FLOAT;
    k INT;
```

```

BEGIN
-----
-- 1. Pick a NORMALIZED query vector
-----
SELECT l2_normalize(page_embedding_norm)
INTO qvec
FROM page
ORDER BY page_id
LIMIT 1;

SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;

DROP TABLE IF EXISTS q9_results;
CREATE TEMP TABLE q9_results(
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
);
-----

-- 2. Ground Truth
-----
DROP TABLE IF EXISTS gt_q9;
CREATE TEMP TABLE gt_q9(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_q9(id, rank)
SELECT page_id,
    ROW_NUMBER() OVER (
        ORDER BY page_embedding_norm <=> qvec, page_id
    )
FROM page
ORDER BY page_embedding_norm <=> qvec, page_id
LIMIT max_k;

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;

```

```
-----  
-- 3. Enable ANN  
-----
```

```
PERFORM set_config('enable_seqscan','off',true);  
PERFORM set_config('enable_indexscan','on',true);
```

```
-----  
-- 4. MAIN LOOP  
-----
```

```
FOREACH k IN ARRAY k_values LOOP
```

```
-----  
-- Indexed part timing  
-----
```

```
idx_start := clock_timestamp();
```

```
DROP TABLE IF EXISTS idx_pages;  
CREATE TEMP TABLE idx_pages(id INT);
```

```
-----  
-- ANN SELECT: top-k pages  
-----
```

```
INSERT INTO idx_pages(id)  
SELECT page_id  
FROM page  
ORDER BY page_embedding_norm <=> qvec, page_id  
LIMIT k;
```

```
-----  
-- Outer aggregation (year, count)  
-----
```

```
PERFORM 1 FROM (  
    SELECT EXTRACT(YEAR FROM r.rev_timestamp::timestamptz) AS year,  
           COUNT(*)  
      FROM revision r  
     JOIN idx_pages p ON r.rev_page = p.id  
    GROUP BY year  
) sub;
```

```
idx_end := clock_timestamp();  
idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;
```

```

-----  

-- Calculate Recall@k  

-----  

SELECT COUNT(*) INTO overlap  

FROM idx_pages i  

JOIN gt_q9 g ON g.id = i.id  

WHERE g.rank <= k;  

recall := overlap::FLOAT / k * 100.0;  

accel := full_ms / idx_ms;  

INSERT INTO q9_results VALUES (k, recall, idx_ms, accel);  

END LOOP;  

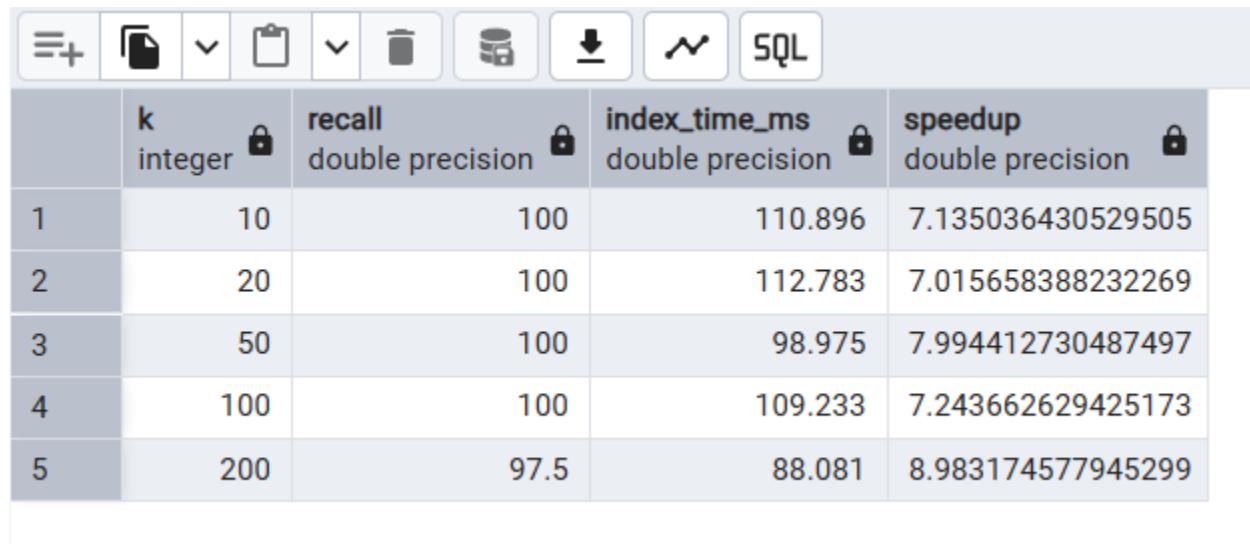
  

END $$;  

SELECT * FROM q9_results ORDER BY k;

```



The screenshot shows a database interface with a toolbar at the top containing various icons for file operations, search, and SQL. Below the toolbar is a table with the following data:

	k integer	recall double precision	index_time_ms double precision	speedup double precision
1	10	100	110.896	7.135036430529505
2	20	100	112.783	7.015658388232269
3	50	100	98.975	7.994412730487497
4	100	100	109.233	7.243662629425173
5	200	97.5	88.081	8.983174577945299

## Q16(IQ1)

```

SET ivfflat.probes = 60;

DO $$  

DECLARE
  k_values  INT[] := ARRAY[10, 20, 50, 100, 200, 500];
  max_k     INT;
  qvec      VECTOR(384);

```

```

offset_val INT := 50; -- you can change this

gt_start TIMESTAMP;
gt_end   TIMESTAMP;
full_ms   FLOAT;

idx_start TIMESTAMP;
idx_end   TIMESTAMP;
idx_ms    FLOAT;

overlap INT;
recall   FLOAT;
accel   FLOAT;

k INT;
BEGIN
-----
-- 1. Pick a NORMALIZED query vector
-----
SELECT l2_normalize(text_embedding_norm)
INTO qvec
FROM text
ORDER BY old_id
LIMIT 1;

SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;

DROP TABLE IF EXISTS iq1_results;
CREATE TEMP TABLE iq1_results(
  k INT,
  recall FLOAT,
  index_time_ms FLOAT,
  speedup FLOAT
);

-----
-- 2. Ground Truth (exact order with offset)
-----
DROP TABLE IF EXISTS gt_iq1;
CREATE TEMP TABLE gt_iq1(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);

```

```

PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_iq1(id, rank)
SELECT old_id,
       ROW_NUMBER() OVER (
           ORDER BY text_embedding_norm <=> qvec, old_id
       )
FROM text
ORDER BY text_embedding_norm <=> qvec, old_id
LIMIT (max_k + offset_val);

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;

```

---

-- 3. Enable ANN

---

```

PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);

```

---

-- 4. MAIN LOOP

---

```

FOREACH k IN ARRAY k_values LOOP

    DROP TABLE IF EXISTS idx_iq1;
    CREATE TEMP TABLE idx_iq1(id INT);

    idx_start := clock_timestamp();

    -- Indexed run with OFFSET
    INSERT INTO idx_iq1(id)
    SELECT old_id
    FROM text
    ORDER BY text_embedding_norm <=> qvec, old_id
    OFFSET offset_val
    LIMIT k;

    idx_end := clock_timestamp();
    idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

```

```

-----
-- Recall computation
-----
SELECT COUNT(*) INTO overlap
FROM idx_iq1 i
JOIN gt_iq1 g ON g.id = i.id
WHERE g.rank > offset_val AND g.rank <= offset_val + k;

recall := overlap::FLOAT / k * 100.0;
accel := full_ms / idx_ms;

INSERT INTO iq1_results VALUES (k, recall, idx_ms, accel);
END LOOP;

END $$;
```

SELECT \* FROM iq1\_results ORDER BY k;

	k integer 	recall double precision 	index_time_ms double precision 	speedup double precision 
1	10	100	64.438	42.05774542971539
2	20	100	38.492	70.40727943468774
3	50	100	40.593	66.76316113615648
4	100	100	38.3	70.76023498694518
5	200	99.5	41.064	65.99739431131893
6	500	98.6	48.552	55.818854012193114

## Q18(IQ3)

```

SET ivfflat.probes = 10;

DO $$
DECLARE
  k_values  INT[] := ARRAY[10, 20, 50, 100, 200];
```

```

max_k    INT;
qvec     VECTOR(384);

len_filter INT := 20000; -- adjust as needed
offset_val INT := 50;   -- OFFSET for IQ3

gt_start  TIMESTAMP;
gt_end    TIMESTAMP;
full_ms   FLOAT;

idx_start TIMESTAMP;
idx_end   TIMESTAMP;
idx_ms    FLOAT;

overlap INT;
recall   FLOAT;
accel   FLOAT;

k INT;
BEGIN
-----
-- 1. Pick a normalized query vector
-----
SELECT l2_normalize(page_embedding_norm)
INTO qvec
FROM page
ORDER BY page_id
LIMIT 1;

-----
-- Prepare max_k
-----
SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;

DROP TABLE IF EXISTS iq3_results;
CREATE TEMP TABLE iq3_results(
  k INT,
  recall FLOAT,
  index_time_ms FLOAT,
  speedup FLOAT
);

```

---

-- 2. Ground Truth (exact full scan with filter + offset)

---

```
DROP TABLE IF EXISTS gt_iq3;
CREATE TEMP TABLE gt_iq3(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_iq3(id, rank)
SELECT page_id,
       ROW_NUMBER() OVER (
           ORDER BY page_embedding_norm <=> qvec, page_id
       )
FROM page
WHERE page_len < len_filter
ORDER BY page_embedding_norm <=> qvec, page_id
LIMIT (max_k + offset_val);

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;
```

---

-- 3. Enable ANN

---

```
PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);
```

---

-- 4. MAIN LOOP

---

```
FOREACH k IN ARRAY k_values LOOP

    DROP TABLE IF EXISTS idx_iq3;
    CREATE TEMP TABLE idx_iq3(id INT);

    idx_start := clock_timestamp();

    -- Indexed ANN + pre-filter + offset + limit

    INSERT INTO idx_iq3(id)
```

```

SELECT page_id
FROM page
WHERE page_len < len_filter
ORDER BY page_embedding_norm <=> qvec, page_id
OFFSET offset_val
LIMIT k;

idx_end := clock_timestamp();
idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

-----
-- Compute Recall@k
-----

SELECT COUNT(*) INTO overlap
FROM idx_iq3 i
JOIN gt_iq3 g ON g.id = i.id
WHERE g.rank > offset_val AND g.rank <= offset_val + k;

recall := overlap::FLOAT / k * 100.0;
accel := full_ms / idx_ms;

INSERT INTO iq3_results VALUES (k, recall, idx_ms, accel);
END LOOP;

END $$;

```

SELECT \* FROM iq3\_results ORDER BY k;

	k integer 	recall double precision 	index_time_ms double precision 	speedup double precision 
1	10	100	7.104	71.41624436936937
2	20	100	7.45	68.09946308724832
3	50	100	8.354	60.73030883409146
4	100	98	6.2	81.8291935483871
5	200	97.5	6.457	78.57224717361004

## Q7 (NQ7)

```
SET ivfflat.probes = 10;

DO $$
DECLARE
  k_values  INT[] := ARRAY[10, 20, 50, 100, 200];
  max_k     INT;
  qvec      VECTOR(384);
  len_filter INT := 20000;

  gt_start  TIMESTAMP;
  gt_end    TIMESTAMP;
  full_ms   FLOAT;

  idx_start TIMESTAMP;
  idx_end   TIMESTAMP;
  idx_ms    FLOAT;

  overlap INT;
  recall   FLOAT;
  accel    FLOAT;

  k INT;
BEGIN
  -----
  -- 1. Pick a normalized query vector
  -----
  SELECT l2_normalize(page_embedding_norm)
  INTO qvec
  FROM page
  ORDER BY page_id
  LIMIT 1;

  -----
  -- Prepare max_k
  -----
  SELECT MAX(x) FROM unnest(k_values) t(x)
  INTO max_k;

  DROP TABLE IF EXISTS q7_results;
  CREATE TEMP TABLE q7_results(
```

```

k INT,
recall FLOAT,
index_time_ms FLOAT,
speedup FLOAT
);

-----
-- 2. Ground Truth (full scan, filter, limit)

DROP TABLE IF EXISTS gt_q7_pages;
CREATE TEMP TABLE gt_q7_pages(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_q7_pages(id, rank)
SELECT page_id,
       ROW_NUMBER() OVER (
           ORDER BY page_embedding_norm <=> qvec, page_id
       )
FROM page
WHERE page_len < len_filter
ORDER BY page_embedding_norm <=> qvec, page_id
LIMIT max_k;

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;

-----
-- 3. ANN Enabled

PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);

-----
-- 4. MAIN LOOP

FOREACH k IN ARRAY k_values LOOP

    DROP TABLE IF EXISTS idx_q7_pages;
    CREATE TEMP TABLE idx_q7_pages(id INT);

```

```

idx_start := clock_timestamp();

-----
-- Indexed inner subquery
-----

INSERT INTO idx_q7_pages(id)
SELECT page_id
FROM page
WHERE page_len < len_filter
ORDER BY page_embedding_norm <= qvec, page_id
LIMIT k;

idx_end := clock_timestamp();
idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

-----
-- Compute Recall@k on inner ANN part
-----

SELECT COUNT(*) INTO overlap
FROM idx_q7_pages i
JOIN gt_q7_pages g ON g.id = i.id
WHERE g.rank <= k;

recall := overlap::FLOAT / k * 100.0;
accel := full_ms / idx_ms;

-----
-- Store result
-----

INSERT INTO q7_results VALUES (k, recall, idx_ms, accel);
END LOOP;

END $$;

SELECT * FROM q7_results ORDER BY k;

```

	k integer 	recall double precision 	index_time_ms double precision 	speedup double precision 
1	10	100	4.075	141.67828220858897
2	20	100	3.854	149.80254281266218
3	50	100	3.794	152.17158671586716
4	100	100	4.298	134.32736156351794
5	200	97.5	5.237	110.24231430208135

## Q20 (IQ5)

```

SET ivfflat.probes = 1;

DO $$

DECLARE
    k_values  INT[] := ARRAY[10, 20, 50, 100, 200, 500];
    max_k      INT;
    qvec      VECTOR(384);
    offset_val INT := 30;

    date_low  TIMESTAMP := '2010-01-01';
    date_high TIMESTAMP := '2015-01-01';

    gt_start  TIMESTAMP;
    gt_end    TIMESTAMP;
    full_ms   FLOAT;

    idx_start TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms    FLOAT;

    overlap INT;
    recall   FLOAT;
    accel   FLOAT;

    k INT;
BEGIN
-----
```

-- 1. Pick normalized query vector

---

```
SELECT l2_normalize(text_embedding_norm)
INTO qvec
FROM text
ORDER BY old_id
LIMIT 1;
```

```
SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;
```

```
DROP TABLE IF EXISTS iq5_results;
CREATE TEMP TABLE iq5_results(
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
);
```

---

-- 2. Ground Truth (exact, no index)

---

```
DROP TABLE IF EXISTS gt_iq5;
CREATE TEMP TABLE gt_iq5(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_iq5(id, rank)
SELECT r.rev_id,
    ROW_NUMBER() OVER (
        ORDER BY (t.text_embedding_norm <=> qvec), t.old_id
    )
FROM text t
JOIN revision r ON t.old_id = r.rev_id
WHERE r.rev_timestamp::timestamp BETWEEN date_low AND date_high
ORDER BY (t.text_embedding_norm <=> qvec), t.old_id
LIMIT (max_k + offset_val);

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;
```

---

```
-- 3. Enable index (ANN)
```

---

```
PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);
```

---

```
-- 4. MAIN LOOP
```

---

```
FOREACH k IN ARRAY k_values LOOP
```

```
    DROP TABLE IF EXISTS idx_iq5;
    CREATE TEMP TABLE idx_iq5(id INT);
```

```
    idx_start := clock_timestamp();
```

```
    INSERT INTO idx_iq5(id)
        SELECT r.rev_id
        FROM text t
        JOIN revision r ON t.old_id = r.rev_id
        WHERE r.rev_timestamp::timestamp BETWEEN date_low AND date_high
        ORDER BY t.text_embedding_norm <=> qvec, t.old_id
        OFFSET offset_val
        LIMIT k;
```

```
    idx_end := clock_timestamp();
    idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;
```

---

```
-- Recall
```

---

```
SELECT COUNT(*) INTO overlap
FROM idx_iq5 i
JOIN gt_iq5 g ON g.id = i.id
WHERE g.rank > offset_val AND g.rank <= offset_val + k;
```

```
recall := overlap::FLOAT / k * 100.0;
accel := full_ms / idx_ms;
```

```

    INSERT INTO iq5_results VALUES (k, recall, idx_ms, accel);
END LOOP;

END $$;

SELECT * FROM iq5_results ORDER BY k;

```

```

10    100    626.84 1.0981717822729884
20    100    585.807     1.1750935034917644
50    100    646.245     1.0651966359507619
100   100    643.461     1.069805318426447
200   100    655.676     1.0498752432603908
500   100    668.43 1.0298430650928296

```

## Q22 (IQ7)

```

SET ivfflat.probes = 10;

DO $$
DECLARE
    k_values  INT[] := ARRAY[10, 20, 50, 100, 200];
    max_k      INT;
    qvec      VECTOR(384);
    len_limit INT := 1000; -- adjust if needed
    offset_val INT := 0;   -- IQ7 has no offset in inner query

    gt_start  TIMESTAMP;
    gt_end    TIMESTAMP;
    full_ms   FLOAT;

    idx_start TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms    FLOAT;

    overlap INT;
    recall   FLOAT;
    accel    FLOAT;

    k INT;
BEGIN

```

---

-- 1. Pick a normalized query vector

---

```
SELECT l2_normalize(page_embedding_norm)
INTO qvec
FROM page
ORDER BY page_id
LIMIT 1;
```

```
SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;
```

```
DROP TABLE IF EXISTS iq7_results;
CREATE TEMP TABLE iq7_results(
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
);
```

---

-- 2. Ground Truth (full exact inner + outer GROUP BY)

---

```
DROP TABLE IF EXISTS gt_iq7_ids;
CREATE TEMP TABLE gt_iq7_ids(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_iq7_ids(id, rank)
SELECT page_id,
       ROW_NUMBER() OVER (ORDER BY (page_embedding_norm <=> qvec), page_id)
FROM page
WHERE page_len < len_limit
ORDER BY (page_embedding_norm <=> qvec), page_id
LIMIT max_k;

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;
```

---

```
-- 3. Enable ANN for indexed run
```

```
-----  
PERFORM set_config('enable_seqscan','off',true);  
PERFORM set_config('enable_indexscan','on',true);  
-----
```

```
-- 4. Main loop for different K
```

```
FOREACH k IN ARRAY k_values LOOP
```

```
DROP TABLE IF EXISTS idx_iq7;  
CREATE TEMP TABLE idx_iq7(id INT);
```

```
idx_start := clock_timestamp();
```

```
INSERT INTO idx_iq7(id)  
SELECT page_id  
FROM page  
WHERE page_len < len_limit  
ORDER BY page_embedding_norm <=> qvec, page_id  
LIMIT k;
```

```
idx_end := clock_timestamp();  
idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;
```

```
-----  
-- Recall computation
```

```
-----  
SELECT COUNT(*) INTO overlap  
FROM idx_iq7 i  
JOIN gt_iq7_ids g ON g.id = i.id  
WHERE g.rank <= k;
```

```
recall := overlap::FLOAT / k * 100.0;  
accel := full_ms / idx_ms;
```

```
INSERT INTO iq7_results VALUES (k, recall, idx_ms, accel);  
END LOOP;
```

```
END $$;
```

```
SELECT * FROM iq7_results ORDER BY k;
```

	k integer 	recall double precision 	index_time_ms double precision 	speedup double precision 
1	10	100	7.287	49.95416495128311
2	20	100	7.199	50.564800666759275
3	50	100	6.609	55.07883189589953
4	100	100	5.396	67.4603409933284
5	200	97	5.697	63.89608565911884

## Q27 (SQ2)

```

SET ivfflat.probes = 50;

DO $$

DECLARE
    -- K values you want to test
    k_values  INT[] := ARRAY[10, 20, 50, 100, 200];
    max_k      INT;

    -- RANGE BUCKETS (WIDE ENOUGH FOR ANN)
    ranges_low  FLOAT[] := ARRAY[0.00, 0.30];
    ranges_high FLOAT[] := ARRAY[0.50, 0.80];

    qvec      VECTOR(384);

    gt_start  TIMESTAMP;
    gt_end    TIMESTAMP;
    full_ms   FLOAT;

    idx_start TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms    FLOAT;

    overlap INT;
    recall   FLOAT;
    accel    FLOAT;

    k INT;
BEGIN

```

---

-- 1. Pick a NORMALIZED Query Vector

---

```
SELECT l2_normalize(text_embedding_norm)
INTO qvec
FROM text
ORDER BY old_id
LIMIT 1;
```

```
SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;
```

```
DROP TABLE IF EXISTS sq2_results;
CREATE TEMP TABLE sq2_results(
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
);
```

---

-- 2. GROUND TRUTH (FULL EXACT SCAN)

---

```
DROP TABLE IF EXISTS gt_sq2;
CREATE TEMP TABLE gt_sq2(id INT PRIMARY KEY, rank INT);
```

```
PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);
```

```
gt_start := clock_timestamp();
```

```
INSERT INTO gt_sq2(id, rank)
SELECT old_id,
    ROW_NUMBER() OVER (
        ORDER BY (text_embedding_norm <=> qvec), old_id
    )
FROM text
WHERE EXISTS (
    SELECT 1
    FROM unnest(ranges_low) rl,
        unnest(ranges_high) rh
    WHERE (text_embedding_norm <=> qvec) BETWEEN rl AND rh
)
ORDER BY (text_embedding_norm <=> qvec), old_id
```

```

LIMIT max_k;

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;

-----
-- 3. ENABLE ANN
-----

PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);

-----
-- 4. MAIN LOOP FOR EACH K
-----

FOREACH k IN ARRAY k_values LOOP

    DROP TABLE IF EXISTS idx_sq2;
    CREATE TEMP TABLE idx_sq2(id INT);

    idx_start := clock_timestamp();

    -----
    -- Indexed candidate pool (LARGE! to avoid recall 0)
    --

    INSERT INTO idx_sq2(id)
    SELECT old_id
    FROM (
        SELECT old_id,
            (text_embedding_norm <= qvec) AS dist
        FROM text
        ORDER BY dist
        LIMIT (max_k * 50)    -- <<< CRITICAL FIX
    ) t
    WHERE EXISTS (
        SELECT 1
        FROM unnest(ranges_low) rl,
            unnest(ranges_high) rh
        WHERE t.dist BETWEEN rl AND rh
    )
    ORDER BY dist, old_id
    LIMIT k;

    idx_end := clock_timestamp();
    idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

```

```

-----
-- 5. COMPUTE RECALL
-----
SELECT COUNT(*) INTO overlap
FROM idx_sq2 i
JOIN gt_sq2 g ON g.id = i.id
WHERE g.rank <= k;

recall := (overlap::FLOAT / k) * 100.0;
accel := full_ms / idx_ms;

INSERT INTO sq2_results VALUES (k, recall, idx_ms, accel);
END LOOP;

END $$;
```

SELECT \* FROM sq2\_results ORDER BY k;

	k integer 	recall double precision 	index_time_ms double precision 	speedup double precision 
1	10	100	38.895	39.327034323177784
2	20	100	36.39	42.034212695795546
3	50	96	42.331	36.13486570125912
4	100	98	35.798	42.72934242136432
5	200	98.5	35.898	42.61031255223132

## Q26 (SQ1)

```

SET ivfflat.probes = 60;

DO $$
DECLARE
    k_values    INT[] := ARRAY[10, 20, 50, 100]; -- r_size = number of ranks
    max_k       INT;

    qvec        VECTOR(384);
    r_positions INT[] := ARRAY[5, 20, 50, 100]; -- example ranks {r1,r2,r3,r4}
```

```

gt_start  TIMESTAMP;
gt_end    TIMESTAMP;
full_ms   FLOAT;

idx_start TIMESTAMP;
idx_end   TIMESTAMP;
idx_ms    FLOAT;

k INT;
BEGIN
-----
-- Pick Query Vector
-----
SELECT l2_normalize(text_embedding_norm)
INTO qvec
FROM text
ORDER BY old_id
LIMIT 1;

SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;

DROP TABLE IF EXISTS sq1_results;
CREATE TEMP TABLE sq1_results(
  k INT,
  result_count INT,
  index_time_ms FLOAT,
  speedup FLOAT
);

-----
-- Ground Truth (full exact ORDER BY + ranking)
-----
DROP TABLE IF EXISTS gt_sq1;
CREATE TEMP TABLE gt_sq1 AS
SELECT old_id,
  ROW_NUMBER() OVER (
    ORDER BY text_embedding_norm <=> qvec, old_id
  ) AS rank
FROM text;

-- Only compute timing of ground truth ranking
gt_start := clock_timestamp();

```

```

PERFORM * FROM gt_sq1 LIMIT 1;

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;

-----
-- MAIN LOOP — Always full scan; no ANN index works here
-----

FOREACH k IN ARRAY k_values LOOP

    idx_start := clock_timestamp();

    -- Extract ranks r1..rk (non-indexable)
    PERFORM 1 FROM (
        SELECT *
        FROM gt_sq1
        WHERE rank IN (SELECT unnest(r_positions))
    ) sub;

    idx_end := clock_timestamp();
    idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

    INSERT INTO sq1_results
    VALUES (k, cardinality(r_positions), idx_ms, full_ms/idx_ms);
END LOOP;

END $$;

SELECT * FROM sq1_results ORDER BY k;

10      4      41.57  0.020110656723598748
20      4      30.508 0.02740264848564311
50      4      27.082 0.030869212022745732
100     4      28.814 0.029013673908516693

```

**Tab 13**

NQ1

```
SET ivfflat.probes = 40;

DO $$
DECLARE
    k_values  INT[] := ARRAY[10, 20, 30, 50, 100, 200];
    max_k      INT;
    qvec       VECTOR(384);

    gt_start   TIMESTAMP;
    gt_end     TIMESTAMP;
    full_ms    FLOAT;

    idx_start  TIMESTAMP;
    idx_end    TIMESTAMP;
    idx_ms     FLOAT;

    overlap    INT;
    recall     FLOAT;
    accel      FLOAT;

    k INT;
BEGIN
-----
-- 1. Pick a NORMALIZED query vector
-----
SELECT l2_normalize(text_embedding_norm)
  INTO qvec
  FROM text
 ORDER BY old_id
 LIMIT 1;

SELECT MAX(x) FROM unnest(k_values) t(x)
  INTO max_k;

DROP TABLE IF EXISTS q1_results;
CREATE TEMP TABLE q1_results(
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
);

```

```
-- 2. Ground Truth (exact full-scan)
-----
DROP TABLE IF EXISTS gt_q1;
CREATE TEMP TABLE gt_q1(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_q1(id, rank)
SELECT old_id,
       ROW_NUMBER() OVER (
           ORDER BY text_embedding_norm <=> qvec, old_id)
FROM text
ORDER BY text_embedding_norm <=> qvec, old_id
LIMIT max_k;

gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;
```

```
-- 3. Enable ANN index
```

```
-----
PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);
```

```
-- 4. MAIN LOOP OVER ALL K VALUES
```

```
-----
FOREACH k IN ARRAY k_values LOOP
```

```
-----  
-- Indexed timing
```

```
-----  
idx_start := clock_timestamp();
```

```
DROP TABLE IF EXISTS idx_q1;
CREATE TEMP TABLE idx_q1(id INT);
```

```
INSERT INTO idx_q1(id)
SELECT old_id
FROM text
```

```

        ORDER BY text_embedding_norm <=> qvec, old_id
        LIMIT k;

        idx_end := clock_timestamp();
        idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

-----
-- Recall@k
-----

SELECT COUNT(*) INTO overlap
FROM idx_q1 i
JOIN gt_q1 g USING(id)
WHERE g.rank <= k;

recall := overlap::FLOAT / k * 100.0;
accel := full_ms / idx_ms;

        INSERT INTO q1_results VALUES (k, recall, idx_ms, accel);
    END LOOP;

END $$;
```

SELECT \* FROM q1\_results ORDER BY k;

NQ11

```

SET ivfflat.probes = 10;

DO $$
DECLARE
    k_values  INT[] := ARRAY[10, 20, 30, 50, 100, 200];
    max_k     INT;
    qvec      VECTOR(384);

    gt_start  TIMESTAMP;
    gt_end    TIMESTAMP;
    full_ms   FLOAT;

    idx_start TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms    FLOAT;

    overlap   INT;
```

```

recall FLOAT;
accel FLOAT;

k INT;
BEGIN
-----
-- 1. Pick a NORMALIZED query vector
-----
SELECT l2_normalize(page_embedding_norm)
INTO qvec
FROM page
ORDER BY page_id
LIMIT 1;

SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;

DROP TABLE IF EXISTS q9_results;
CREATE TEMP TABLE q9_results(
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
);
-----

-- 2. Ground Truth
-----
DROP TABLE IF EXISTS gt_q9;
CREATE TEMP TABLE gt_q9(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_q9(id, rank)
SELECT page_id,
    ROW_NUMBER() OVER (
        ORDER BY page_embedding_norm <=> qvec, page_id
    )
FROM page
ORDER BY page_embedding_norm <=> qvec, page_id
LIMIT max_k;

```

```
gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;
```

---

```
-- 3. Enable ANN
```

---

```
PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);
```

---

```
-- 4. MAIN LOOP
```

---

```
FOREACH k IN ARRAY k_values LOOP
```

---

```
-- Indexed part timing
```

---

```
idx_start := clock_timestamp();
```

```
DROP TABLE IF EXISTS idx_pages;
CREATE TEMP TABLE idx_pages(id INT);
```

---

```
-- ANN SELECT: top-k pages
```

---

```
INSERT INTO idx_pages(id)
SELECT page_id
FROM page
ORDER BY page_embedding_norm <=> qvec, page_id
LIMIT k;
```

---

```
-- Outer aggregation (year, count)
```

---

```
PERFORM 1 FROM (
    SELECT EXTRACT(YEAR FROM r.rev_timestamp::timestamptz) AS year,
           COUNT(*)
    FROM revision r
    JOIN idx_pages p ON r.rev_page = p.id
    GROUP BY year
) sub;
```

```

idx_end := clock_timestamp();
idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;

-----
-- Calculate Recall@k
-----
SELECT COUNT(*) INTO overlap
FROM idx_pages i
JOIN gt_q9 g ON g.id = i.id
WHERE g.rank <= k;

recall := overlap::FLOAT / k * 100.0;
accel := full_ms / idx_ms;

INSERT INTO q9_results VALUES (k, recall, idx_ms, accel);
END LOOP;

END $$;

```

SELECT \* FROM q9\_results ORDER BY k;

IQ1

SET ivfflat.probes = 20;

```

DO $$

DECLARE
    k_values  INT[] := ARRAY[10, 20, 30, 50, 100, 200];
    max_k     INT;
    qvec      VECTOR(384);

    offset_val INT := 50; -- you can change this

    gt_start  TIMESTAMP;
    gt_end    TIMESTAMP;
    full_ms   FLOAT;

    idx_start TIMESTAMP;
    idx_end   TIMESTAMP;
    idx_ms    FLOAT;

    overlap INT;

```

```

recall FLOAT;
accel FLOAT;

k INT;
BEGIN
-----
-- 1. Pick a NORMALIZED query vector
-----
SELECT l2_normalize(text_embedding_norm)
INTO qvec
FROM text
ORDER BY old_id
LIMIT 1;

SELECT MAX(x) FROM unnest(k_values) t(x)
INTO max_k;

DROP TABLE IF EXISTS iq1_results;
CREATE TEMP TABLE iq1_results(
    k INT,
    recall FLOAT,
    index_time_ms FLOAT,
    speedup FLOAT
);

-----
-- 2. Ground Truth (exact order with offset)
-----
DROP TABLE IF EXISTS gt_iq1;
CREATE TEMP TABLE gt_iq1(id INT PRIMARY KEY, rank INT);

PERFORM set_config('enable_seqscan','on',true);
PERFORM set_config('enable_indexscan','off',true);

gt_start := clock_timestamp();

INSERT INTO gt_iq1(id, rank)
SELECT old_id,
    ROW_NUMBER() OVER (
        ORDER BY text_embedding_norm <=> qvec, old_id
    )
FROM text
ORDER BY text_embedding_norm <=> qvec, old_id
LIMIT (max_k + offset_val);

```

```
gt_end := clock_timestamp();
full_ms := EXTRACT(EPOCH FROM (gt_end - gt_start)) * 1000;
```

---

```
-- 3. Enable ANN
```

---

```
PERFORM set_config('enable_seqscan','off',true);
PERFORM set_config('enable_indexscan','on',true);
```

---

```
-- 4. MAIN LOOP
```

---

```
FOREACH k IN ARRAY k_values LOOP
```

```
    DROP TABLE IF EXISTS idx_iq1;
    CREATE TEMP TABLE idx_iq1(id INT);
```

```
    idx_start := clock_timestamp();
```

```
    -- Indexed run with OFFSET
    INSERT INTO idx_iq1(id)
    SELECT old_id
    FROM text
    ORDER BY text_embedding_norm <=> qvec, old_id
    OFFSET offset_val
    LIMIT k;
```

```
    idx_end := clock_timestamp();
    idx_ms := EXTRACT(EPOCH FROM (idx_end - idx_start)) * 1000;
```

---

```
-- Recall computation
```

---

```
    SELECT COUNT(*) INTO overlap
    FROM idx_iq1 i
    JOIN gt_iq1 g ON g.id = i.id
    WHERE g.rank > offset_val AND g.rank <= offset_val + k;
```

```
    recall := overlap::FLOAT / k * 100.0;
    accel := full_ms / idx_ms;
```

```
    INSERT INTO iq1_results VALUES (k, recall, idx_ms, accel);
END LOOP;

END $$;

SELECT * FROM iq1_results ORDER BY k;
```