Optimizing Index Structures to Support Semantic Queries in Relational Databases

Victor Kamel & Dr. Sean Chester

University of Victoria, Department of Computer Science, March 12, 2023

1. Introduction

Computers are not able to natively understand text. Thus, when text is stored in a computer system, it is represented as "strings" of characters encoded using a standard such as ASCII or UTF-8. This makes certain queries based on the structure of the text straightforward.
For example:

- The query "Find all places in Canada that start with *Eagle*" may yield:
  - Eagle Lake First Nation, Ontario
  - Eagle River, Ontario
  - Eaglesham, Alberta

Semantic Queries

- What if we used natural language processing to allow database queries that incorporate the *meaning* of the text?
- Can this be done efficiently?

```
SELECT * FROM T
WHERE City LIKE "Seattle" - "US" + "Canada"
LIMIT 3
---
1. Vancouver, British Columbia
2. Kelowna, British Columbia
3. Calgary, Alberta
```

The objective of this research is to investigate the practicality of using semantic identifiers for strings that enable semantic queries to be performed.

2. Word Embeddings & Similarity

What do we mean when we say two words are semantically similar?

Word Embeddings

- Word embeddings are N-dimensional, real-valued vector representations of words in some vocabulary V. The embeddings of words that are used in *similar contexts* appear in proximity to each other in $\mathbb{R}^N$.
- Context is determined by the words that surround the given word in the training corpus.
- The vectors are generated using the hidden layer of a unsupervised machine learning model.
- Vector arithmetic can be performed on word embeddings: *vector("King")* − *vector("Man")* + *vector("Woman")* creates a vector that is closest to the embedding of the word *"Queen"* [2].

Cosine Similarity

- Given two word embeddings $\vec{a}, \vec{b} \in \mathbb{R}^N$, we need a method to measure their similarity. A common metric used for this purpose is based on the cosine of the angle between them:

$$S_{cos}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} = \frac{\sum_{i=1}^{N} a_i b_i}{\sqrt{\sum_{i=1}^{N} a_i^2}\sqrt{\sum_{i=1}^{N} b_i^2}}.$$

Indices & Top-K Queries

- Since word embeddings can have hundreds of dimensions [2], in order to store them efficiently as identifiers, we need to reduce the number of dimensions using a technique such as *principal component analysis* (PCA).
- Here, we consider primarily the 2-dimensional case.
- In order to make queries efficient, an index structure optimized for cosine similarity is created.
- We will focus on top-K queries: Given a set $V \subseteq \mathbb{R}^2$, $|V| = n$ we define the top-K query based on cosine similarity to a given $\vec{q} \in \mathbb{R}^2$ with $k \in [0, n]$ as

$$TopK(V, \vec{q}, k) = \text{argmax}_{K \subseteq V, |K|=k} \sum_{\vec{p} \in K} S_{cos}(\vec{p}, \vec{q}).$$

3. Linear Scans

Algorithm 1: *HeapScan(V, $\hat{q}$, k)*: Return top k most similar vectors to $\hat{q}$ in V.
input : Array of normalized vectors V, normalized vector $\hat{q}$ and # of vectors to return k.
output: Array of k (similarity, vector) tuples.

```
PQ ← MakeMinHeap(V[0 : k])
foreach p̂ ∈ V[k :] do
    PQ.push((p̂ · q̂, p̂))
    PQ.pop()
return Elements of PQ
```

In Algorithm 1, S_{cos} is calculated between every vector in V and $\hat{q}$, and the results is inserted into a heap of size k. This algorithm runs in $O(n \log k)$ time with $O(k)$ extra space.

Algorithm 2: *IntroScan(V, $\hat{q}$, k)*: Return top k most similar vectors to $\hat{q}$ in V.
input : Array of normalized vectors V, normalized vector $\hat{q}$ and # of vectors to return k.
output: Array of k (similarity, vector) tuples.

```
Initialize array K of size |V|
for i ← 0 to |V| − 1 do K[i] ← (V[i] · q̂, V[i])
Introselect(K, k)
return K[0 : k]
```

In Algorithm 2, S_{cos} is calculated between every vector in V and $\hat{q}$ as before, but next we use the *introselect* algorithm which can partition the array such that the top k elements appear in the first k positions in $O(n)$ expected time [3] and $O(n)$ space.

Both algorithms are simple to implement, and generalize well to higher dimensions.

4. Spatial Partitioning

Algorithm 3: *UniformSpatialIndex(S, $\hat{q}$, k)*: Return top k most similar vectors to $\hat{q}$ in V.
input : Spatial index S, normalized vector $\hat{q}$ and number of vectors to return k.
output: Array of k vectors.

```
if |V| ≤ k then return V
bucketPQ ← MakeMinHeap(S.buckets) // Based on distance to q̂
pointPQ ← MakeMaxHeap()
while (pointPQ.size < k or
  D_cos(q̂, pointPQ.peek()) >
  D_cos(q̂, bucketPQ.peek())) do
    bucket ← bucketPQ.pop()
    for v⃗ ∈ S.getVectors(bucket) do
        if pointPQ.size() < k then pointPQ.push(v⃗)
        else
            if D_cos q̂, (v⃗) < pointPQ.peek() then
                pointPQ.pop()
                pointPQ.push(v⃗)
return Elements of pointPQ
```

In Algorithm 3, a uniform partitioning is used as seen in Figure 1. We can use a modification of the Distance Browsing algorithm [1] in order to perform a BFS starting at the bucket that would contain $\hat{q}$. Figure 2 visualizes the distance function D_{cos} between $\hat{q}$ and any given bucket.

The total runtime is $O(b \log b + n \log n) \approx O(n \log n)$ (where $b << n$ is the number of buckets) and $O(b + n)$ extra space is required. However, we expect that for an average query where k is much smaller than n, only a portion of the buckets will be checked. There is also an initial overhead to create the index, and this algorithm is more difficult to generalize to higher dimensions.

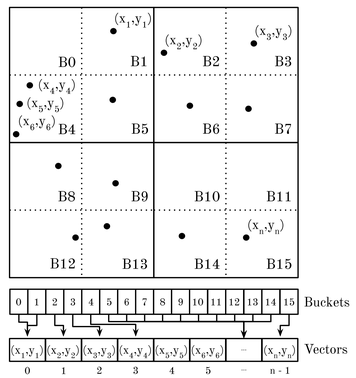

Figure 1. Uniform Spatial Partitioning.

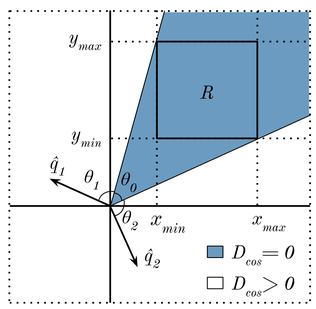

Figure 2. Distance function between axis aligned bounding box R and vector $\hat{q}$.

5. Radial Partitioning

Algorithm 4: *RadialIndex(S, $\hat{q}$, k)*: Return top k most similar vectors to $\hat{q}$ in V.
input : Array S of normalized vectors sorted by angle, normalized $\hat{q}$ and # of vectors to return k.
output: Array of k vectors.

```
if |V| ≤ k then return V
else if |V| = 0 or k = 0 then return Array()
// Find index of vector p̂ closest to q̂
p ← BinarySearch(S, q̂)
K ← Array(S[p])
n ← S.size()
if k ≠ 1 then
    left ← (p − 1)%n
    right ← (p + 1)%n
    while K.size() < k do
        l ← S[left]
        r ← S[right]
        if D_cos(q̂, l) < D_cos(q̂, r) then
            K.append(l)
            left ← (left − 1)%n
        else
            K.append(r)
            right ← (right + 1)%n
return K
```

1. Find closest point.
2. Search in both directions.

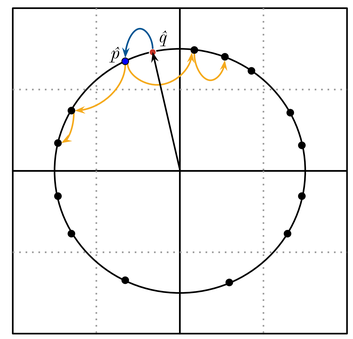

Figure 3. Diagram of Radial Spatial Partitioning top-K algorithm, $k = 5$.

If we normalize each vector and store them in a sorted order based on their angle, we no longer need the Distance Browsing algorithm and can search only the k closest points to $\hat{q}$ directly, as Figure 3 and Algorithm 4 illustrate. The overall runtime complexity is $O(k + \log n)$, with an $O(1)$ extra space complexity.

6. Results

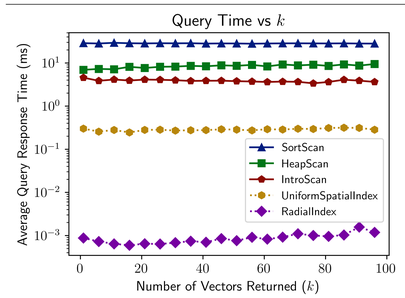

Figure 4. Varying k.

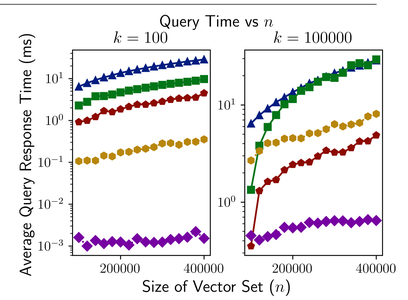

Figure 5. Varying n.

In order to measure the relative performance of these data structures, the query response time was measured against k and n. The algorithms were be implemented in C++, and run on an x86 machine. The vectors used were pre-trained word embeddings from [4], dimensionally reduced to two dimensions using PCA. From Figures 4 and 5, we can see that the measured runtime characteristics agree with the theoretical values. Clearly, the radial partitioning approach is superior.

7. References

[1] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, jun 1999.

[2] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia, June 2013. Association for Computational Linguistics.

[3] David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exp.*, 27(8):983–993, 1997.

[4] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

8. Acknowledgments

This research was supported by the Jamie Cassels Undergraduate Research Awards, University of Victoria, and supervised by Dr. Sean Chester.