

# Algoritmos de vizinhos mais próximos aproximados

Disciplina: TCC00288

Professor: Luiz André Portes Paes Leme

Barbara Keren Nascimento C. Guarino

Bruna de Assunção Santos

Gabriel Gavazzi Felix

Gabriel Ramalho Braga

Tatiana Machado Brito dos Santos

Vitor Balestro Dias da Silva



# 1. Descrição do problema

Informalmente, o problema dos  $k$  vizinhos mais próximos consiste em encontrar os  $k$  vetores de um dado conjunto  $S$  que estão mais próximos de um dado vetor  $v$ . Formalmente, sejam  $k, n \in \mathbb{N}$ ,  $S \subseteq \mathbb{R}^n$  e  $v \in \mathbb{R}^n$ . Assuma que  $S$  é um conjunto finito com  $m$  elementos, onde  $m \geq k$ , e seja  $d$  uma distância em  $\mathbb{R}^n$ . Considere a ordenação

$$S = \{v_1, \dots, v_m\}$$

onde  $d(v_j, v) \geq d(v_i, v)$  sempre que  $j > i$  (isto é,  $S$  está ordenado pela distâncias a  $v$ ). Desejamos encontrar o conjunto

$$\text{NN}_k(v, S, d) := \{v_1, \dots, v_k\}.$$

Não é difícil implementar um algoritmo computacional que forneça uma solução exata para este problema. Entretanto, a depender do tamanho de  $S$ , tal solução seria computacionalmente cara. De fato, é preciso calcular as distâncias dos elementos de  $S$  a  $v$  e, em seguida, ordená-las.

Desta forma, o desafio é desenvolver e implementar algoritmos que encontrem *aproximadamente* os  $k$  vizinhos mais próximos. Em linhas gerais, o objetivo é obter *boas* aproximações com *baixo* custo de execução. Note que os adjetivos em itálico referem-se a propriedades que são, ao menos em alguma medida, subjetivas.

Em uma descrição de alto nível, um *algoritmo de vizinhos mais próximos aproximados* (vamos chamá-lo de  $\pi$ ) consiste em duas etapas:

- (i) uma fase de *pré-processamento*, em que são construídas estruturas de dados para  $S$ ;
- (ii) a fase de processamento em que, dados um número  $k \in \mathbb{N}$  e um vetor  $v$ , é retornado um conjunto  $\text{ANN}_k(v, S, d) = \{w_1, \dots, w_{k^*}\} \subseteq S$  (com  $k \leq k^*$ ) de vizinhos mais próximos aproximados de  $v$ .

Em linhas gerais, a qualidade da aproximação obtida é mensurada comparando o conjunto aproximado obtido  $\text{ANN}_k(\pi, v, S, d)$  com a solução exata (ordenada)  $\text{NN}(v, S, d) = \{v_1, \dots, v_k\}$ . Formalmente, definimos o *recall* da solução aproximada como

$$\text{recall}(\pi) = \frac{\left| \left\{ w \in \text{ANN}_k(\pi, v, S, d) : d(w, v) \leq d(w, v_k) \right\} \right|}{k},$$

onde denotamos por  $|A|$  a quantidade de elementos do conjunto  $A$ . Nesta definição, estamos contando quantos vetores da solução aproximada têm distância para  $v$  mais próxima do que  $v_k$ , que é o último vetor da solução exata ordenada.

Note que para calcular o recall, precisamos apenas de  $d(w, v_k)$ . Assim, esta distância será chamada *distância de referência* a partir de agora.

## 2. Tabelas

Consideramos o problema sobre um conjunto  $S$  de 1000000 de vetores em  $\mathbb{R}^{128}$  com  $k = 100$ . No banco de dados, temos as seguintes tabelas:

- (i) **object**: contém os vetores do conjunto  $S$ . Portanto, é uma tabela com 1000000 de linhas, em que cada linha é um vetor de 128 coordenadas;

- (ii) **tquery**: contém um conjunto de 10000 casos de teste, para os quais a solução exata é conhecida ;
- (iii) **neighbor**: contém as soluções exatas para os vetores da tabela **tquery**. Assim, é uma tabela de 10000 linhas em que cada linha é um vetor de 100 coordenadas.

Para o que segue, denote por  $\text{tabela}[j]$  a  $j$ -ésima linha de uma tabela, e por  $\text{vetor}[j]$  a  $j$ -ésima coordenada de um vetor. Para cada  $1 \leq j \leq 10000$ , o vetor  $\text{neighbor}[j]$  contém os *índices* dos 100 vetores da tabela **object** que estão (ordenadamente) mais próximos o vetor  $\text{tquery}[j]$ . Estes índices se referem à tabela **object**. Por exemplo, se

$$\text{neighbor}[10] = (20, 12, 14, \dots),$$

então os vetores de **object** mais próximos do vetor  $\text{tquery}[10]$  são

$$\text{object}[20], \text{object}[12], \text{object}[14], \dots$$

Para a conveniência do leitor, o esquema completo (com os identificadores das colunas) está descrito no Apêndice [A.1](#).

### 3. Pré-processamento

Na fase de pré-processamento, primeiro executamos um algoritmo de *clusterização* de  $S$ . Para este fim, usamos o algoritmo *KMeans* da biblioteca *scikit* da linguagem *Python*. Informalmente, este algoritmo agrupa os vetores de  $S$  em  $k$  conjuntos de vetores próximos entre si, e retorna os *centróides* destes conjuntos (pense no centróide como um vetor que é, de alguma forma, central dentro do conjunto). Consideramos  $k = 128$ , e isso significa que o algoritmo retornará uma tabela com 128 vetores de  $S$  (os *centróides*). Estes vetores estão armazenados na tabela **sight**. Esta tabela tem duas colunas: **id**, que é a chave primária (inteiros sequenciais) e **centroid**, que armazena o centróide correspondente (vetor de inteiros).

O próximo passo é criar uma tabela que informe o centróide mais próximo de cada vetor da tabela **object** (e a distância do vetor ao seu centróide mais próximo). Assim, dado um vetor **query**, podemos, por exemplo, restringir a busca aos vetores cujo centróide mais próximo é o centróide mais próximo de **query**. A tabela que guarda estas informações é chamada **closer\_centroids**, e o código para gerá-la é exibido abaixo. Vale notar que esta etapa é custosa. Note que a chamada da função recebe como parâmetros a quantidade de vetores de **object** para os quais o centróide mais próximo será calculado e um valor de *offset*. Para evitar esgotamento de memória principal, dividimos a tabela rodando a função para cada uma de suas partes. Em um notebook de categoria intermediária, cada execução para 100000 vetores levou em torno de 4 minutos.

```

1 CREATE TYPE tuple AS (ind int, dist double precision);
2
3 CREATE OR REPLACE FUNCTION get_closer_centroids(qtd int, offs int) RETURNS void AS
4   $$
5   DECLARE
6     object_line record;
7     sight_line record;
8     distances_vector tuple[];
9     current_tuple tuple;
10    dist double precision;
11    min_dist double precision;

```

```

11     i int;
12     ind int;
13 BEGIN
14     FOR object_line IN SELECT * FROM object ORDER BY id LIMIT qtd OFFSET offs LOOP
15         FOR sight_line IN SELECT * FROM sight_ LOOP
16             dist = euclidean_distance(object_line.features,sight_line.centroid);
17             current_tuple = (sight_line.id,dist);
18             distances_vector[sight_line.id] = current_tuple;
19         END LOOP;
20         min_dist = distances_vector[1].dist;
21         ind = 1;
22         FOR i in 1..128 LOOP
23             IF (distances_vector[i].dist < min_dist) THEN
24                 min_dist = distances_vector[i].dist;
25                 ind = distances_vector[i].ind;
26             END IF;
27         END LOOP;
28         INSERT INTO closer_centroids VALUES (object_line.id,ind, min_dist);
29     END LOOP;
30 END;
31 $$ LANGUAGE plpgsql;
32

```

Listing 1: Obtendo a tabela de centróides mais próximos

O esquema lógico das tabelas criadas na etapa de pré-processamento está no Apêndice [A.2](#)

## 4. Método 1: busca por regiões parametrizadas

Este método de busca dos 100 vetores mais próximos de um vetor  $\mathbf{q}$  é como segue:

- **Passo 1:** encontramos o centróide  $\mathbf{c}$  mais próximo de  $\mathbf{q}$ ;
- **Passo 2:** computamos a distância  $d$  de  $\mathbf{q}$  para  $\mathbf{c}$ ;
- **Passo 3:** escolhemos parâmetros  $0 < \alpha < 1$ ,  $\beta > 1$  e  $0 < \gamma < 1$  e, dentre os vetores  $v \in S$  (tabela **object**) cujo centróide mais próximo é  $\mathbf{c}$ , encontramos aqueles que satisfazem:

(i)  $\alpha d < d(v, \mathbf{c}) < \beta d$ , e

(ii)  $\gamma < \cos(v - \mathbf{c}, \mathbf{q} - \mathbf{c}) < 1$ .

Isto é, consideramos a interseção entre a coroa circular com centro em  $\mathbf{c}$  e raios  $\alpha d$  e  $\beta d$  (interno e externo, respectivamente) com o cone de vetores  $w$  tais que o cosseno do ângulo entre  $w - \mathbf{c}$  e  $\mathbf{q} - \mathbf{c}$  é menor do que  $\gamma$ . Veja a Figura [4.1](#) para uma ilustração da região considerada.

Para fins de simplificar a notação, a partir de agora vamos denotar por  $S_{\mathbf{c}}$  o subconjunto dos vetores de  $S$  que têm  $\mathbf{c}$  como centróide mais próximo.

Aqui, vale um comentário sobre a eficiência deste algoritmo. A tabela **closer\_centroids** já fornece a distância entre cada vetor  $v \in S_{\mathbf{c}}$  cujo centróide  $\mathbf{c}$ . Assim, nenhuma distância será calculada no passo 3. Mais ainda, lembre-se de que o valor do cosseno entre  $v - \mathbf{c}$  e  $\mathbf{q} - \mathbf{c}$  é computado pela

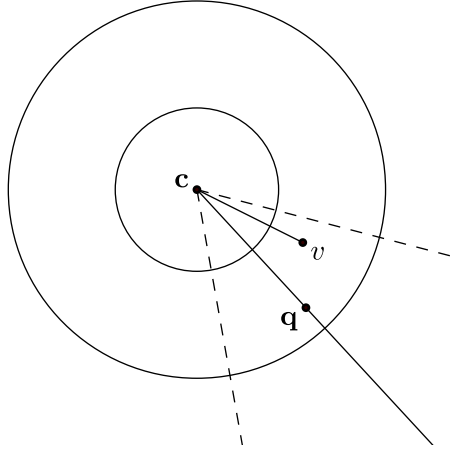


Figura 4.1: Ilustração em duas dimensões do método 1.

fórmula

$$\cos(v - \mathbf{c}, \mathbf{q} - \mathbf{c}) = \frac{\langle v - \mathbf{c}, \mathbf{q} - \mathbf{c} \rangle}{\|v - \mathbf{c}\| \|\mathbf{q} - \mathbf{c}\|} = \frac{\langle v - \mathbf{c}, \mathbf{q} - \mathbf{c} \rangle}{d(v, c) \cdot d}$$

A norma de  $v - \mathbf{c}$  é igual à distância entre  $v$  e  $\mathbf{c}$  e, portanto, não precisará ser calculada. A norma de  $\mathbf{q} - \mathbf{c}$  é igual à  $d$ , que foi calculado no passo 1. Daí, apenas o produto interno deverá ser computado. Ou seja, para cada  $v \in S_{\mathbf{c}}$ , o produto interno  $\langle v - \mathbf{c}, \mathbf{q} - \mathbf{c} \rangle$  será calculado. Para calcular cada um destes produtos internos, são realizadas 128 multiplicações dentro de um *loop* que atualiza uma soma.

*Observação 4.1.* Note que para encontrar o conjunto  $S_{\mathbf{c}}$ , teremos que executar a seguinte consulta:

```

1
2 SELECT * FROM closer_centroids WHERE closer_centroid_ind = id(c);
3
```

Listing 2: Encontrando  $S_{\mathbf{c}}$

onde  $\text{id}(\mathbf{c})$  é a chave do centróide  $\mathbf{c}$  (na tabela **sight**, é claro). Para otimizar estas consultas, criamos um índice na tabela **closer\_centróides**:

```

1
2 CREATE INDEX closer_centroid_idx ON closer_centroids (closer_centroid_ind);
3
```

Listing 3: Índice na chave do centróide mais próximo

Esta consulta, que é feita originalmente sobre uma tabela com 1000000 de entradas, leva 60 milissegundos, em média (novamente, em um notebook pessoal de nível intermediário).

Vamos agora à implementação do método. Primeiro, notamos que o método depende de três parâmetros. Ao início da execução, esta tripla de parâmetros será armazenada em uma tabela (**method1\_parameters\_key**). Desta forma, cada tripla de parâmetros pode ser referenciada por uma chave primária. Se a tripla de parâmetros já estiver cadastrada, então ela não será recadastrada.

Os resultados encontrados (isto é, os 100 vetores mais próximos aproximados) serão armazenados na tabela **result\_table\_method1**). Cada linha desta tabela armazenará a chave da tripla de parâmetros considerada, o índice do vetor **query**, o vetor resultado e a distância do vetor resultado para o vetor **query** (esta distância será usada para computar o *recall* mais tarde). Se a tabela já contiver resultados para a chave (tripla de parâmetros, vetor query) considerados, a execução será abortada (para evitar duplicidade).

O código segue abaixo:

```

1 CREATE OR REPLACE FUNCTION ann_method1(query_vector_id int, alpha double
    precision, beta double precision, gamma double precision)
2 RETURNS void AS $$
3 DECLARE
4     query_vector int[]; -- q
5     closer_centroid int[]; -- c
6     diff_centroid_query int[]; -- q - c
7     diff_candidate_centroid int[]; -- v - c
8     closer_centroid_index int;
9     candidate_vector int[]; -- v
10    dist_query_centroid double precision; -- ||q-c|| = d
11    dist_candidate_centroid double precision; -- ||v-c||;
12    dist_candidate_query double precision; -- ||v-q||;
13    inner_product double precision;
14    cosine double precision;
15    counter int;
16    n int;
17    parameters_key_value int;
18    reference_dist double precision;
19    line_ record;
20    i int;
21 BEGIN
22     n = (SELECT COUNT(*) FROM method1_parameters_key AS par WHERE par.alpha_ =
        alpha AND par.beta_ = beta AND par.gamma_ = gamma);
23     IF (n = 0) THEN
24         INSERT INTO method1_parameters_key (alpha_,beta_,gamma_) VALUES (alpha,beta
        ,gamma);
25     END IF;
26     parameters_key_value = (SELECT par.id FROM method1_parameters_key AS par WHERE
        par.alpha_ = alpha AND par.beta_ = beta AND par.gamma_ = gamma);
27     n = (SELECT COUNT(*) FROM result_table_method1 WHERE parameter_key_value =
        parameters_key_value AND query_vector_id_ = query_vector_id);
28     IF (n != 0) THEN
29         RAISE EXCEPTION 'The function was already computed for this query vector
        and these parameter values';
30     END IF;
31
32     SELECT query INTO query_vector FROM tquery WHERE id = query_vector_id;
33     closer_centroid_index = get_closer_centroid_tquery(query_vector_id);
34     SELECT centroid INTO closer_centroid FROM sight_ WHERE id =
        closer_centroid_index;
35     dist_query_centroid = euclidean_distance(closer_centroid,query_vector);
36     FOR i IN 1..128 LOOP
37         diff_centroid_query[i] = query_vector[i] - closer_centroid[i];
38     END LOOP;
39     counter = 0;
40     FOR line_ IN SELECT * FROM closer_centroids WHERE closer_centroid_ind =
        closer_centroid_index LOOP

```

```

41     SELECT features INTO candidate_vector FROM object WHERE id = line_.id;
42     dist_candidate_centroid = euclidean_distance(candidate_vector,
closer_centroid);
43     dist_candidate_query = euclidean_distance(candidate_vector, query_vector);
44     FOR i IN 1..128 LOOP
45         diff_candidate_centroid[i] = candidate_vector[i] - closer_centroid[i];
46     END LOOP;
47     IF (dist_candidate_centroid > alpha * dist_query_centroid AND
dist_candidate_centroid < beta * dist_query_centroid) THEN
48         inner_product = get_inner_product(diff_candidate_query,
diff_candidate_centroid) :: double precision;
49         cosine = inner_product/(dist_candidate_centroid * dist_query_centroid);
50         IF (cosine > gamma AND cosine < 1) THEN
51             INSERT INTO result_table_method1(parameter_key_value,query_vector_id_
,vector,distance) VALUES
52                 (parameters_key_value,query_vector_id,candidate_vector,
dist_candidate_query);
53             counter = counter + 1;
54         END IF;
55     END IF;
56 END LOOP;
57 IF counter > 100 THEN
58     reference_dist = (SELECT distance FROM result_table_method1 AS r
59     WHERE r.parameter_key_value = parameters_key_value AND
60         r.query_vector_id_ = query_vector_id
61     ORDER BY distance
62     LIMIT 1
63     OFFSET 99);
64
65     DELETE FROM result_table_method1 AS r
66     WHERE r.parameter_key_value = parameters_key_value AND
67         r.query_vector_id_ = query_vector_id AND
68         distance > reference_dist;
69 END IF;
70
71 END;
72 $$ LANGUAGE plpgsql;
73

```

Listing 4: O algoritmo do Método 1.

## 5. Calculando o recall

Para calcular o *recall* de uma determinada execução, primeiro obtemos a distância referência correspondente (através da tabela **neighbor**). Usamos a função explicitada abaixo para calcular a distância de referência do  $j$ -ésimo vetor da tabela **object** (isto é, o vetor cujo id é  $j$ ).

```

1 CREATE OR REPLACE FUNCTION get_reference_distance(j int) RETURNS double precision
  AS
2 $$
3 DECLARE
4     query_vector int[];
5     neighbors_vector int[];
6     index_ int;

```

```

7   reference_vector double precision [];
8
9 BEGIN
10  query_vector := (SELECT query FROM tquery WHERE id = j);
11  neighbors_vector := (SELECT neighbors FROM neighbors WHERE id = j);
12  index_ := neighbors_vector[100];
13  reference_vector := (SELECT features FROM object WHERE id = index_);
14
15  RETURN euclidean_distance(query_vector,reference_vector);
16
17 END;
18 $$ LANGUAGE plpgsql;
19

```

Listing 5: Calculando a distância de referência.

O algoritmo exato para computar o *recall* depende do método adotado. Para o Método 1, a tabela de resultados (**result\_table\_method1** já fornece a distância de cada vetor da solução ao vetor **query**. Assim, usamos a seguinte função:

```

1 CREATE OR REPLACE FUNCTION get_recall_method1(query_vector_index int,
2   parameters_key int) RETURNS double precision AS
3 $$
4 DECLARE
5   i int;
6   query_vector int[];
7   result_vector double precision[];
8   reference_distance double precision;
9   hit_count int;
10  dist double precision;
11  line_ record;
12  recall_ double precision;
13 BEGIN
14  hit_count = 0;
15  reference_distance = get_reference_distance(query_vector_index);
16  FOR line_ IN (SELECT * FROM result_table_method1 WHERE parameter_key_value =
17    parameters_key AND query_vector_id_ = query_vector_index) LOOP
18    IF(line_.distance <= reference_distance) THEN
19      hit_count = hit_count + 1;
20    END IF;
21  END LOOP;
22  recall_ = hit_count :: double precision / 100;
23  INSERT INTO recall_table_method1(query_vector_id,parameters_key_,recall)
24  VALUES
25    (query_vector_index,parameters_key,recall_);
26  RETURN recall_;
27 END;
28 $$ LANGUAGE plpgsql;
29

```

Listing 6: Calculando o *recall* no Método 1.

Observe que esta função calcula o *recall* obtido para determinado vetor (da tabela **tquery**) com um determinado conjunto de parâmetros ( $\alpha$ ,  $\beta$  e  $\gamma$ ). Assim, podemos rodar o algoritmo do Método 1 para diversos vetores de **tquery** e diversas combinações de parâmetros antes de computar o *recall*



de cada combinação. Isto torna mais conveniente obter estatísticas de eficiência do método, como ficará claro mais tarde.

Para computar e armazenar o *recall* de uma amostra de 100 vetores de **tquery** para determinado conjunto de parâmetros, executamos a função:

```

1 CREATE OR REPLACE FUNCTION get_recall_for_all(parameters_key int) RETURNS void AS
2 $$
3 DECLARE
4     alpha double precision;
5     beta double precision;
6     gamma double precision;
7 BEGIN
8     alpha = (SELECT alpha_ FROM method1_parameters_key WHERE id = parameters_key);
9     beta = (SELECT beta_ FROM method1_parameters_key WHERE id = parameters_key);
10    gamma = (SELECT gamma_ FROM method1_parameters_key WHERE id = parameters_key);
11    FOR i IN 1..100 LOOP
12        PERFORM ann_method1(i,alpha,beta,gamma);
13        PERFORM get_recall_method1(i,parameters_key);
14    END LOOP;
15 END;
16 $$ LANGUAGE plpgsql;
17

```

Listing 7: Computando o *recall* para todos os vetores de **tquery**.

Diversas tabelas foram criadas para armazenar os resultados das consultas e os *recalls* obtidos. Para fins de conveniência do leitor, adicionamos o esquema lógico das tabelas do Método 1 no Apêndice [A.3](#).

## 6. Discussão dos resultados

A tabela abaixo compara o Método 1 com o algoritmo de força bruta. Os parâmetros de cada execução do Método 1 estão informados entre parênteses. As colunas **Média**, **Mínimo**, **Máximo** e **Desvio padrão** referem-se aos valores de *recall* obtidos para os 100 primeiros vetores da tabela **tquery**. A coluna **Tempo médio** refere-se ao tempo médio de execução do método para um vetor específico informado como parâmetro.

	Média	Mínimo	Máximo	Desvio padrão	Tempo médio
Método 1 (0.01; 1000; 0.33)	0.531	0.05	1	0.249	0.760 s
Método 1 (0.01; 100; 0.33)	0.531	0.05	1	0.248	0.621 s
Método 1 (0.5; 30; 0.4)	0.493	0.04	1	0.252	0.538 s
Método 1 (0.5; 10; 0.5)	0.252	0	1	0.239	0.415 s
Força bruta	1	1	1	0	25 s

Note que o método de força bruta é mais de 40 vezes mais lento do que o Método 1 com parâmetros 0.5, 30 e 0.4 (que tem *recall* próximo de 50%). O código do método de força bruta encontra-se abaixo:

```

1 CREATE OR REPLACE FUNCTION ann_brute_force(query_index int) RETURNS void AS $$
2 DECLARE
3     query_vector int[];

```

```

4      object_vector int[];
5      line_ record;
6      dist double precision;
7 BEGIN
8      SELECT query INTO query_vector FROM tquery WHERE id = query_index;
9      FOR line_ IN (SELECT * FROM object) LOOP
10         object_vector = line_.features;
11         dist = euclidean_distance(object_vector,query_vector);
12         INSERT INTO result_table_brute_force(query_vector_ind, vector_index,
distance) VALUES
13             (query_index,line_.id,dist);
14     END LOOP;
15
16     dist = (SELECT distance FROM result_table_brute_force ORDER BY distance LIMIT
1 OFFSET 99);
17     DELETE FROM result_table_brute_force WHERE query_vector_ind = query_index AND
distance > dist;
18
19 END;
20 $$ LANGUAGE plpgsql;
21

```

Listing 8: O método de força bruta.

## 7. Referências

- [1] M. Aumüller, E. Bernhardsson, A. Faithfull: *ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms*. Disponível em: <https://arxiv.org/pdf/1807.05614.pdf>

## A. Appêndice

### A.1. Esquema do Dataset SIFT-128

```

1 CREATE TABLE object
2     (id INT NOT NULL PRIMARY KEY,
3      features int[128]
4     );
5
6 CREATE TABLE tquery
7     (id INT NOT NULL PRIMARY KEY,
8      query int[128]
9     );
10
11 CREATE TABLE neighbors
12     (id INT NOT NULL PRIMARY KEY,
13      neighbors int[100],
14     );
15

```

Listing 9: O esquema do pré-processamento

## A.2. Esquema das estruturas de dados do pré-processamento

```
1 CREATE TABLE sight_  
2   (id INT NOT NULL PRIMARY KEY,  
3     centroid int[],  
4   );  
5  
6 CREATE TABLE closer_centroids  
7   (id INT NOT NULL,  
8     closer_centroid_ind INT,  
9     dist double precision  
10  );
```

Listing 10: O esquema do pré-processamento

## A.3. Esquema do Método 1

```
1 CREATE TABLE method1_parameters_key  
2   (id SERIAL PRIMARY KEY,  
3     alpha_ double precision,  
4     beta_ double precision,  
5     gamma_ double precision  
6   );  
7  
8 CREATE TABLE result_table_method1  
9   (id SERIAL PRIMARY KEY,  
10  parameter_key_value int,  
11  query_vector_id_ int,  
12  vector int[],  
13  distance double precision,  
14  FOREIGN KEY (parameter_key_value) REFERENCES method1_parameters_key(id),  
15  FOREIGN KEY (query_vector_id_) REFERENCES object(id)  
16  );  
17  
18 CREATE TABLE recall_table_method1  
19   (id SERIAL PRIMARY KEY,  
20  query_vector_id int,  
21  parameters_key_ int,  
22  recall double precision,  
23  FOREIGN KEY (query_vector_id) REFERENCES tquery(id),  
24  FOREIGN KEY (parameters_key) REFERENCES method1_parameters_key(id)  
25  );  
26
```

Listing 11: O esquema do Método 1