T. Grust et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2019),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 225

Fast Approximated Nearest Neighbor Joins For Relational Database Systems

Michael Günther[1] Maik Thiele[1] Wolfgang Lehner[1]

Abstract: K nearest neighbor search (kNN-Search) is a universal data processing technique and a fundamental operation for word embeddings trained by word2vec or related approaches. The benefits of operations on dense vectors like word embeddings for analytical functionalities of RDBMSs motivate an integration of kNN-Joins. However, kNN-Search, as well as kNN-Joins, have barely been integrated into relational database systems so far. In this paper, we develop an index structure for approximated kNN-Joins working well on high-dimensional data and provide an integration into PostgreSQL. The novel index structure is efficient for different cardinalities of the involved join partners. An evaluation of the system based on applications on word embeddings shows the benefits of such an integrated kNN-Join operation and the performance of the proposed approach.

Keywords: approximated nearest neighbor search, product quantization, RDBMS, word embeddings

1 Introduction

Word embedding techniques are powerful to study the syntactic and semantic relations between words by representing them in dense vectors. By applying algebraic operations on these vectors, semantic relationships such as word analogies, gender-inflections, or geographical relationships can be easily recovered [LG14]. Due to the powerful capabilities of word embeddings, some recent papers proposed their integration into relational databases [BBS17, Gü18]. This allows exploiting external knowledge during query processing by comparing terms occurring in a database schema with terms stored in word embeddings. To give some examples: a user may query all products in a product database and rank them according to their mean similarity to terms like "allergen" or "sensitizer". In the context of a movie database, a kNN-Search can be performed to return the top-3 nearest neighbors of each movie title (see q_1 in Fig. 1). Given the movie "Godfather" as input this might result in "1972" (the release year), "Scarface" (another popular movie in the same genre) and "Coppola" (the director). Our main observation is that most of these SQL database word embedding operations perform similarity search with *k nearest neighbor search (kNN)* as a common subtask. Furthermore, the domain of the kNN-Search often needs to be restricted to terms that also appear in the database relation, i.e. the domain modeled by the database schema. In this way arbitrary terms, having their origins in the

[1] Technische Universität Dresden, Institut for Systems Architecture, Dresden Database Systems Group, Nöthnitzer Straße 46, 01187 Dresden, firstname.lastname@tu-dresden.de

doi:10.18420/btw2019-15

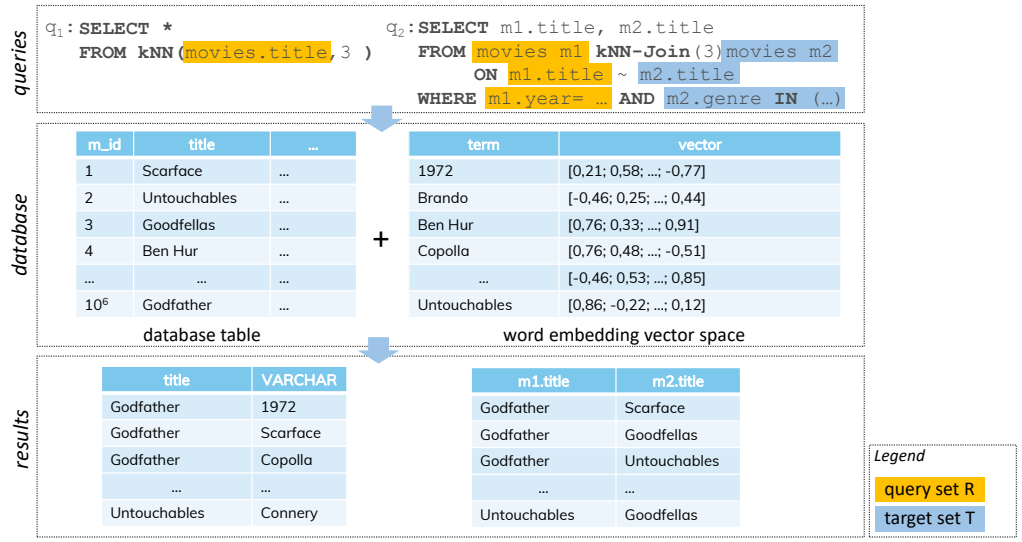


Fig. 1: Two Example Queries: kNN-Search and kNN-Join

large text corpora on which the word embedding models have been trained on, are filtered out. To give an example: if a kNN-Search is performed on an attribute *movie titles* the user usually expects to get the most similar movies but not release years, actors, directors or other terms that do not relate to the movie domain at all. Technically this boils down to a *k nearest neighbor join* (kNN-Join) that combines each element in a *query set R* with the k elements in a *target set T* that are closest to it. This is shown by q_2 in Fig. 1 that extends q_1 by a target set containing just movie titles and that returns movie titles only. Due to the high dimensionality of word embeddings (100 to 300 dimensions are a typical number) and large input data sets (query and target set), the kNN-Join is an extremely expensive operation. Therefore, we investigate approximation and indexing techniques based on vector quantization approaches, especially product quantization [JDS11], to accelerate kNN-Joins in the realm of RDBMSs. In particular, our contributions are the following:

- We identify two different kNN-Join query types with different needs regarding the supporting index structures.
- We propose a novel index structure which can cope with both query types and is flexible enough to deliver good performance on them.
- We detail how to efficiently process an approximated kNN-Join query that adapts to different query and target set sizes.
- We provide a practical implementation of the operator and our optimizations in PostgreSQL, which allows us to meaningfully evaluate the operator using high-dimensional data and fully-featured SQL queries regarding both, accuracy and runtime.

The remainder of the paper is structured as follows: In the next section, we define the kNN-Join problem and derive the challenges which arise by supporting this operation on high-dimensional data. In Sect. 3, we provide the fundamentals of the vector quantization techniques which form the foundation for our proposed index structure. In Sect. 4, we present our inverted product quantization index as well as our approximated and adaptive kNN-Join implementation. Given two real-world datasets, we evaluate the accuracy and response time of this novel operator regarding different input relation sizes in Sect. 5. Finally, we survey the related work in Sect. 6 and conclude the paper in Sect. 7.

2 Problem Description

Given two vector sets R and T in a d-dimensional Euclidean space $\mathbb{R}^d$ and two elements $\mathbf{r}$ and $\mathbf{t}$, with $\mathbf{r} \in R$ called query vectors and $\mathbf{t} \in T$ called target vectors, a kNN query is defined as follows:

Definition 2.1. The kNN query of $\mathbf{r}$ over T, noted $kNN(\mathbf{r}, T)$, can be defined as:
$kNN(\mathbf{r}, T) = \arg\min_{\{\mathbf{t_1},\ldots,\mathbf{t_k}\} \in T^{[k]}} \sum_{i=1}^{k} d(\mathbf{r}, \mathbf{t_i})$.

Here d denotes the distance function between two elements. Typically, in the context of word embeddings, the cosine distance is used. However, in case of normalized vectors $\mathbf{r}$ and $\mathbf{t}$, the cosine distance is proportional to the squared Euclidean distance. The normalization of the vectors does not change the cosine distance. Thus the $kNN(\mathbf{r}, T)$ for any $\mathbf{r}$ and T can be computed using both metrics. If the query is not just one element but instead a set, the operation is denoted as kNN-Join.

Definition 2.2. The kNN-Join between a query set R and a target set T is defined as:
$kNN(R \ltimes T) = \{\langle \mathbf{r}, \mathbf{t} \rangle | \mathbf{t} \in kNN(\mathbf{r}, T), \mathbf{r} \in R\}$.

Challenges The aim of this paper is to provide a kNN-Join which is particularly suitable for high-dimensional data and varying target sets. In detail, we identify the following challenges:

1. Batchwise execution of large query sets: In contrast to a simple kNN-Search, it must be possible to execute large amounts of nearest neighbor queries at once for kNN-Joins. Most of the approximated kNN-Search (ANN-Search) approaches assume that the set of target vectors contains a very large number of vectors, however, they do not process large amounts of query vectors. In the case of kNN-Joins, the number of queries can be much larger than the target vector set.
2. High-dimensional data: Previous work on kNN-Joins for relational database systems focuses mostly on low-dimensional data [YLK10]. Because of the *curse of dimensionality*, the distances of pairs of sample vectors from a high-dimensional vector space tend to differ only little [Be99]. Therefore, techniques for exact kNN-Joins, trying to hierarchical

partition vector spaces, cannot be applied efficiently. Hence, the system must support suitable approximated search techniques to handle large vector sets.

3. Adaptive kNN-Join algorithm: An index for the kNN-Join stores all possible target vectors. However, a target set T often contains just a small subset of all vectors in the index. For example: target set for q_2 in Fig. 1 only contains vectors of movies published in a specific year out of millions of other vectors. The kNN-Join algorithm therefore must be adaptive to different target set sizes and should enable fast approximated search. With respect to the cardinality of R and T we identified two different kNN-Join query types in a database system:

kNN queries with small query set R and large target set T: This is the ordinary type of kNN queries, which most of the kNN frameworks assume.

kNN queries with large query set R and small target set T: This case is rather specific to the use in database systems and is currently not supported.

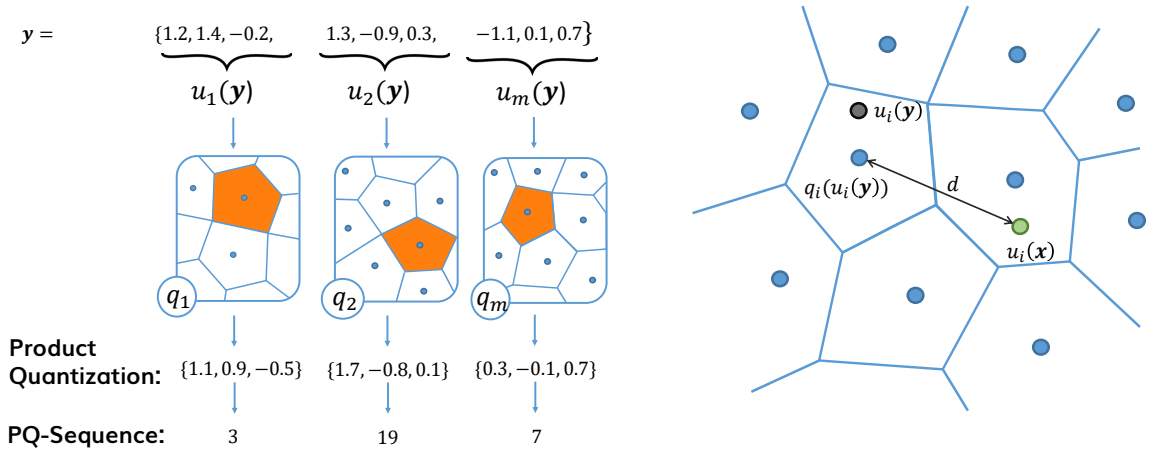
4. Different demands on precision and response time: Regarding the approximation of the vector similarity, it might be relevant for a user to specify how strongly the approximated nearest neighbors should correspond with the exact values. On the contrary, real-world systems need to comply with certain latency constraints, e.g., for exploratory data processing fast response times are crucial. Consequently, the approximated kNN-Join should provide features to configure such trade-offs. Providing this tunable trade-offs would also support query execution in an online aggregation manner, i.e., get estimates of a kNN-Join query as soon as the query is issued and steadily refine during its execution.

3 Vector Quantization

The index structure we propose is based on different *vector quantization* techniques. *Vector quantization* is able to transform vectorial data in an approximated compact representation [Gr84]. Furthermore, it enables fast approximated distance calculation which subsequently can be used to speed up kNN-Join operations. It is the basis of product quantization as well as the basis for inverted indexing techniques described in Sect. 3.2 and Sect. 3.3.

3.1 Quantization Function

Vector quantization can be implemented by a quantization function $q(\mathbf{y})$ which assigns a vector $\mathbf{y}$ to a centroid $\mathbf{c_j} \in C$ where $\mathbf{c_j}$ is the vector of C which has the lowest distance to $\mathbf{y}$. There are different ways to obtain such a quantization function, which is specified by the centroid set C and a distance function d. As a distance function, we use the *Euclidean distance*. The set C should be selected so that the distortion is minimal. The *k-means* algorithm is commonly used to achieve this goal for a given number of centroids $|C|$. An approximated representation of a vector dataset can be obtained by replacing every vector $\mathbf{y} \in Y$ (floating point values) with their centroid id j of its quantization centroid $\mathbf{c_j} = q(\mathbf{y})$ (integer values).



(a) Generating PQ sequences (b) Asymmetric Distance Calculation

Fig. 2: Product Quantization

3.2 Product Quantization

A simple vector quantization approach might lead to a quite inaccurate representation of the vector dataset. For a more precise representation, huge numbers of centroids would be necessary that are impossible to process or even to store. For this reason, *product quantization* [JDS11] applies multiple quantizers on m subvectors $u_1(\mathbf{y}), \ldots, u_m(\mathbf{y})$ of the original vector $\mathbf{y}$ (see Fig. 2a.). Those quantizers are defined by quantization functions $q_1, \ldots, q_m$ with $q_i : \mathbb{R}^d \rightarrow C_i$. Typically, the cardinalities $|C_1|, \ldots, |C_m|$ are equal. The product quantization is the sequence of centroids obtained by that process.

$$\underbrace{y_1, \ldots y_d}_{u_1(\mathbf{y})}, \ldots, \underbrace{y_{(D-d)+1}, \ldots y_D}_{u_m(\mathbf{y})} \rightarrow q_1(u_1(\mathbf{y})), \ldots, q_m(u_m(\mathbf{y})) \tag{1}$$

Using a dictionary denoted as codebook, the sequence of centroid vectors can be compactly represented as a sequence of centroid ids.

kNN-Search with PQ-Index Product quantization sequences can be utilized to accelerate the calculation of nearest neighbors by providing a fast way to compute approximated squared distances. Approximated square distances between a query vector $\mathbf{x}$ and a vector $\mathbf{y}$ for which a product quantization sequence is available can be calculated by Eq. (2).

$$\hat{d}^2(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{m} d(u_i(\mathbf{x}), q_i(u_i(\mathbf{y})))^2 \tag{2}$$

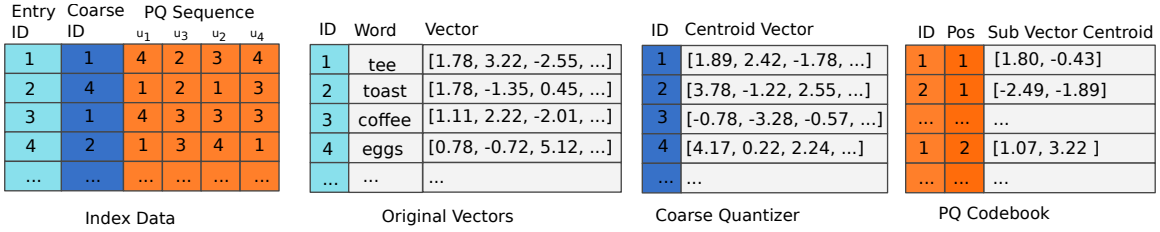


Fig. 3: Index Data Structure

The squared distances $d(u_i(\mathbf{x}), q_i(u_i(\mathbf{y})))^2$ have to be precomputed at the beginning of the search process. For every subvector $u_i(\mathbf{x})$ there are $|C_i|$ distance values to calculate, since $q_i(u_i(\mathbf{y}))$ can be any value of C_i. The distance measure is denoted as asymmetric by [JDS11], since it is defined between quantized and non-quantized vectors as visualized in Fig. 2b. Despite the effort of the preprocessing the technique reduces the computational costs, since the number of index entries in large vector datasets is much higher than $m \cdot |C_i|$, the number of those squared distances. Furthermore, the compact representation makes it easier to provide fast access to the index entries which can also improve performance.

3.3 Inverted Index Structures for Approximated Nearest Neighbor Search

For the standard product quantization search, it is necessary to calculate $|T|$ distance values for every query vector against the target set T. To reduce the number of distance computations and achieve *non-exhaustive* search behavior one can divide the dataset into partitions of vectors called cells which are locally close to each other. After that, only vectors which are in the same cell as the query vector are considered as candidates for the nearest neighbors. One can also extend the search to a certain amount of nearby cells. There are several ways to define the cells: typically, vector quantization is employed by [JDS11] to build the so-called IVFADC index. Here, a cell is defined by the subspace which the quantization function assigns to the same centroid (*Voronoi cell*). In [BBS17] it is stated that either LSH or k-means is used for that. Babenko et al. use product quantization [BL12] to build a fine granular inverted index which is described in detail in Sect. 4.3.
However, these non-exhaustive methods prohibit the search in arbitrary subsets of the index entries which is needed for smaller target sets T. To give an example: if one queries only in a cell of the vector space and the target set is small it is very likely that the index might return an empty set of candidates. To solve this problem we propose an adaptive kNN-Join algorithm which determines a suitable number of cells and provides multiple lookups.

4 Adaptive Search Algorithm for Approximated kNN-Joins

We propose an adaptive search algorithm for kNN-Joins which can cope with arbitrary target sets T. The index structure used by this algorithm is described in Sect. 4.1 and the

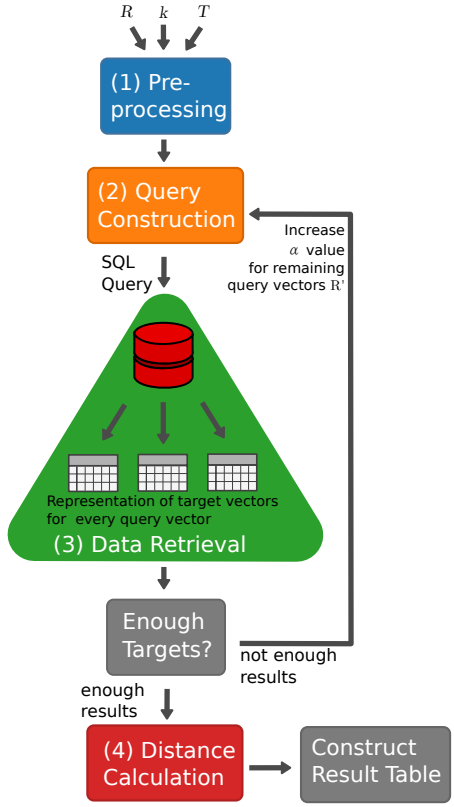
algorithm itself in Sect. 4.2. We employ the inverted multi-index described in Sect. 4.3 that is especially efficient for RDBMSs and propose an approach to estimate the number of targets read out from the inverted index in Sect. 4.3 which is necessary to use it for different target sets. The distance computation is outlined in Sect. 4.4. This is based on product quantization as described in Sect. 3.2. However, for the adaptation to different cardinalities of T, we employ a modification described in Sect. 4.5. Further optimizations are presented in Sect. 4.6.

4.1 Index structure

The data structure of our proposed index for word vectors is shown in Fig. 3. Every index entry in the `Index Data` table has an id to reference it (`Entry ID`). In addition, it consists of a `Coarse ID` referring to a partition the vector belongs to for the inverted search and a product quantization sequence (`PQ Sequence`). Every partition has a centroid which is stored in the `Coarse Quantizer` table. The centroids of subvectors for the product quantization are stored in a `Codebook`. Each of those PQ centroids has an `ID` which corresponds to codes in the product quantization sequences and a position `Pos` $\in \{1, \ldots m\}$ which refers to the position of the subvector it is calculated for $(u_1, \ldots, u_m)$. The original vectors are stored in a `Original Vectors` table.

4.2 Adaptive kNN-Join Algorithm

Fig. 4a shows a flow chart and Fig. 4b the pseudo code of our proposed kNN-Join algorithm. As input parameters, the algorithm gets a set of query vectors $R = \mathbf{r_1}, \ldots \mathbf{r_n}$, the set of target vectors T represented as subset of index entry ids and the desired number k of nearest neighbors. Furthermore, there are two configuration parameters: α and Th_{flex}. The value α determines the minimum number of targets per result that has to be considered for the search process. A higher value of α leads to a higher precision of the result set. The value Th_{flex} configures the calculation of distances with the product quantization which is discussed in detail in Sect. 4.5. The algorithm consists of four steps: At first, there is a *preprocessing step* (Line 3 to 6), which is necessary for the product quantization based distance calculation described in Sect. 3.2. There are two different types of distance calculations via product quantization based on either `SHORT_CODES` or `LONG_CODES`. The first one is suitable for large numbers of distance calculations per query whereas the second one is applicable for fewer distance calculations. Details are provided in Sect. 4.5. Th_{flex} determines the limit of distance calculation where the algorithm switches from `LONG_CODES` to `SHORT_CODES`, whereas the distance calculations depends on $\alpha \cdot k$. The precomputed distance values of subvectors are stored in $\mathcal{D}_{pre}$. In the *query construction step*, the retrieval of database entries from the inverted index is prepared. This involves the calculation of the coarse quantization C for every query vector $\mathbf{r_i}$ in Line 9 which returns a sequence of the coarse centroid ids from the `Coarse Quantizer` table (see Fig. 3) in decreasing order according to the distance between



(a) Flow Chart of Algorithm

Require:
Selectivity: α
Threshold Flexible-PQ: Th_{flex}

1: **function** ADAPTIVE-KNN-JOIN(R, k, T, α)
2: $R' \leftarrow R, j \leftarrow 1$
3: **if** $\alpha \cdot k > Th_{flex}$ **then** ▷ only for product quantization
4: $\mathcal{D}_{pre} \leftarrow$PREPROCESSING($R$, T, SHORT_CODES)
5: **else**
6: $\mathcal{D}_{pre} \leftarrow$PREPROCESSING($R$, T, LONG_CODES)
7: **while** $R' \neq \emptyset$ **do**
8: **for all** $\mathbf{r_i} \in R$ **do**
9: $C^* \leftarrow$COARSEQUANTIZE($\mathbf{r_i}$)
10: $\omega \leftarrow$ESTIMATEORDER($C, T, \alpha \cdot k \cdot j$)
11: $centr(i) \leftarrow \{c \in C \mid order(c) < \omega\}$
12: $query \leftarrow$CONSTRUCTQUERY($centr, T$)
13: $T_{sub} \leftarrow$EXECUTE($query$)
14: $R' \leftarrow \{\mathbf{r_i} \mid |T_{sub}(i)| < \alpha \cdot k\}$
15: $j \leftarrow 2 \cdot j$
16: **for all** $\mathbf{r_i} \in R$ **do**
17: **for all** $\mathbf{t} \in T_{sub}(i)$ **do**
18: $d \leftarrow$DISTFUNC($\mathbf{r_i}$, t, $\mathcal{D}_{pr*e}$)
19: UPDATE($top_k[\mathbf{r_i}]$, d)
20: **return** top_k

* ordered list of centroid ids

(b) Pseudo-Code of the Algorithm

Fig. 4: Adaptive kNN-Join Algorithm

the coarse centroid and the query vector $\mathbf{r_i}$. Every centroid id corresponds to a partition in the index. The number of partitions ω to be considered is estimated by the ESTIMATEORDER function (details are given in Sect. 4.3). Based on ω and C the set of the centroids to be retrieved from the index for $\mathbf{r_i}$ is then added to $centr(i)$. After that, a single SQL query is constructed to retrieve the data for all query vectors from the index (*data retrieval step*, Line 13). Then, the query vectors R' for which not enough index entries could be retrieved are determined. For them, another query construction and retrieval iteration is done with a less conservative order estimation (modified by j). If the number of targets is sufficient, the distance values between every query vector $\mathbf{r_i}$ and its respective index entries $T_{sub}(i)$ are calculated by a distance function DISTFUNC ((*distance calculation step*). We elaborate more on the distance function in Sect. 4.4. The best candidates for the *kNN* operation are held in a sorted list top_k which is updated after every distance calculation.

4.3 Inverted Multi-Index and Partition Estimation

In Sect. 3.3 we described the advantages of inverted indexing for ANN. However, inverted indexing in general is poorly suitable when the target set T is only a subset of all vectors

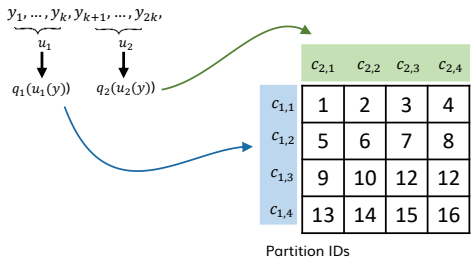


Fig. 5: Inverted Multi Index

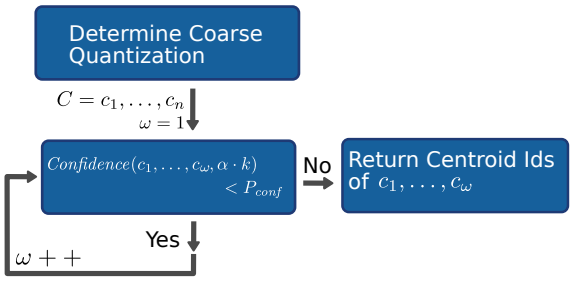


Fig. 6: Confidence Estimation

in the index T_I, i.e. $T \ll T_I$. For this case it is not obvious how many partitions should be considered. To solve this problem, we propose a method to estimate the number of targets observed by the search when a certain number of partitions is read out from the index. This allows us to pick the optimal number of partitions (Fig. 4b Line 11). To optimize the coarse quantization step (Fig. 4b Line 9) we use an inverted multi-index [BL12] which enables fast lookups even if large numbers of index partitions are needed.

Inverted Multi-Index A simple inverted index based on quantization could be created by clustering all possible target vectors T_I into n distinct partitions $P_1 \dot{\cup} \ldots \dot{\cup} P_n$ which corresponds to the Voronoi cells of the centroids $\mathbf{c_1}, \ldots, \mathbf{c_n}$. To determine the partitions in which to search for a query $\mathbf{r}$, one has to calculate all the distances $d(\mathbf{r}, \mathbf{c_1}), \ldots, d(\mathbf{r}, \mathbf{c_n})$. However, this could be time-consuming for database queries with a large query set R. To solve this problem, [BL12] propose to use product quantization to obtain more smaller clusters for the partitions with only a few centroids. Suppose the product quantization sequences which serve as labels for the partitions consist of two centroid indexes $c_1, c_2 \in \{1, \ldots, n\}$, there are n^2 numbers of partitions (see Fig. 5). But, to determine the nearest clusters one has to calculate only the $2 \cdot n$ square distances between the subvectors of the query vector centroids stored in a codebook. Subsequently, an order of centroids in accordance with the distances to the query vector can be obtained by using the algorithm described in [BL12]. The data structure shown in Fig. 3 is designed for a simple coarse quantizer. If product quantization according to the inverted multi-index is used, the coarse quantization table is replaced by a second codebook relation and the ids c_1 and c_2 are represented by a single id $id_c = c_1 \cdot n + c_2$ in `Coarse ID`.

Estimation of the Number of Targets The overall objective of the estimation (Fig. 4b Line 10) is to determine a suitable number $\omega \leq n$ of nearest partitions in a way, that the probability P_{est} that it is necessary to run further database requests for the query vector $\mathbf{r_i}$ is lower than a certain value $1 - P_{conf}$. This is done by iteratively incrementing ω until the confidence value obtained by a probabilistic model is higher than P_{conf} (see Fig. 6). The estimation relies on statistics about the distribution of the index. Those contain the

relative sizes of all partitions $P_1, \ldots, P_n$ compared to the whole index size (the total number of vectors). For the estimation, we consider the set of all index entries T_I, the target set of the current query T_i and a set of partitions $P_1 \dot{\cup} \ldots \dot{\cup} P_\omega = T_p$ which are selected as the partitions with the nearest centroids to the query vector. We then want to estimate the probability $1 - P_{est}$ that T_i contains at least $\beta = k \cdot \alpha$ entries which corresponds to the condition in the algorithm of Fig. 4b in Line 14. It corresponds to the cardinality of $T_i \cup T_p$. For this purpose, we leverage a hypergeometric probability distribution (Eq. (3)) which describes the probability to get s successes by drawing M elements out of a set of N elements without replacement. In our case, s is the desired number of targets in $T_i \cup T_p$, M is the cardinality of T_p and N equals $|T_I|$. The probability of drawing at least β target vectors from the set T_I of all vectors in the index can be calculated with Eq. (4) by using the cumulative distribution function.

$$h(X = s \;;\; |T_I|, |T_i|, |T_p|) = \frac{\binom{|T_p|}{s}\binom{|T_I|-|T_p|}{|T_i|-s}}{\binom{|T_I|}{|T_i|}} \tag{3}$$

$$\mu = |T_i| \cdot \frac{|T_p|}{|T_I|} \quad \sigma^2 = |T_i| \cdot \frac{|T_p|}{|T_I|} \cdot \Big(1 - \frac{|T_p|}{|T_I|}\Big) \cdot \frac{|T_I| - |T_i|}{|T_I| - 1}$$

$$1 - P_{est} = h_{cdf}(\beta - 1 \;;\; |T_I|, |T_i|, |T_p|) = 1 - \sum_{s=0}^{\beta-1} \frac{\binom{|T_p|}{s}\binom{|T_I|-|T_p|}{|T_i|-s}}{\binom{|T_I|}{|T_i|}} \tag{4}$$

However, because of the complexity of the computation of the binomial coefficients, we have to use an approximation based on the normal distribution Eq. (5). To obtain the approximation, we use the mean μ and the variance σ^2 from the hypergeometric distribution (see Eq. (3)). In contrast to Eq. (3), which involves a cumulative distribution function, here, the number k is a *specific* number of targets which corresponds to $T_i \cup T_p$.

$$N(k; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\Big[-\frac{(k-\mu)^2}{2\sigma^2}\Big] \tag{5}$$

The approximation of the probability of getting at least $\beta - 1$ targets (Eq. (6)) is then obtained by its cumulative distribution function. The addition of 0.5 serves as a continuity error correction. It is added to the formula since the hypergeometric distribution is a discrete probability distribution. Since β is an integer value, $k < \beta - 1$ corresponds to $k < \beta - 0.5$.

$$\begin{aligned} h_{cdf}(\beta - 1; \mu, \sigma^2) &\approx 1 - \sum_{s=0}^{\beta-1} N(s; \mu, \sigma^2) \\ &\approx 1 - \frac{1}{2} \cdot \left(1 + erf\Big(\frac{(\beta - 1) + 0.5 - \mu}{\sqrt{2}\sigma}\Big)\right) \end{aligned} \tag{6}$$

The probability h_{cdf} of getting enough targets can be increased by raising the number of partitions in T_p which corresponds to the coarse order ω in the algorithm in Line 10 of Fig. 4b. The algorithm chooses ω in a way that it is minimal and h_{cdf} is higher than a certain probability P_{conf} which is also termed as the *confidence value*. From experimental results, we noticed that 0.8 seems to be a good value to achieve high response time for the algorithm.

4.4 Distance Calculation

Beside the adaptive number of partitions and vectors to be considered by the kNN-Join algorithm, the trade-off between precision and runtime also depends on the distance function, namely: *(1) exact calculation*, *(2) product quantization* and *(3) product quantization with post verification*. Product quantization is the fastest one but also provides the lowest precision. It calculates distance values as described in Sect. 3.2. The exact calculation is too slow, especially for a large number of target vectors and a large α. Method (3) strikes a balance between both extremes and therefore represents the default distance function. In the first place it calculates the approximated distance values using product quantization for $k \cdot \alpha$ targets. Second, it refines the $k \cdot pvf$ best candidates with the exact method to obtain the final top-k. Here, the post-verification factor pvf is the major factor which influences the precision of the kNN computation. By adjusting it the user can control the trade-off between precision and runtime as desired in Challenge 3 of Sect. 2. For further details see the evaluation in Sect. 5.2.

4.5 Flexible Product Quantization

The product quantization index provides two parameters: the number of subvectors m and the number of centroids per quantizer $|C|$. The optimal setting of both parameters depends on the desired precision and response time as well as on the typical number of distance calculations $\alpha \cdot k$ which are performed for every query vector. In general, higher values of m and $|C|$ correspond to higher precision and higher response times.
If the target set size $\alpha \cdot k$ is large, the computation of the distances (Line 18) is the most time-consuming step whereas for small target sets, the computation time of the preprocessing step (Line 3) becomes more and more prevalent. Since a low value for m, i.e. a low number of subvectors, corresponds to a faster distance calculation, the product quantization speed for large target sets depends mainly on m. However, with a decreasing number of vectors $\alpha \cdot k$, the preprocessing step (Line 3) has also a high computational effort which is mainly influenced by $|C|$. A decreasing number $|C|$ corresponds to a faster search process.
To be efficient in both situations, we introduce a flexible product quantization search procedure. For product quantization search with a small number of distance calculations $\alpha \cdot k < Th_{flex}$ an index is created with a large number of subvectors $m = 2 \cdot m'$ but only a small number of centroids $|C|$ (see 4). This is called the `LONG_CODES` mode, since the pq sequences consist of a larger number m of ids. For a larger number of distance calculations,

the number of distances to sum up for each distance calculation (see Eq. (2)) can be reduced by precalculating squared distances for pairs of centroids $\langle \mathbf{c_j}, \mathbf{c_{j+1}} \rangle$ and pairs of subvectors $\langle u_j(\mathbf{r}), u_{j+1}(\mathbf{r}) \rangle$ (Sect. 4.2):

$$d(\langle u_j(\mathbf{r}), u_{j+1}(\mathbf{r}) \rangle, \langle \mathbf{c_j}, \mathbf{c_{j+1}} \rangle)^2 = d(u_j(\mathbf{r}), q_j(u_j(\mathbf{y})))^2 + d(u_j(\mathbf{r}), q_{j+1}(u_{j+1}(\mathbf{y})))^2 \quad (7)$$
$$\textit{where}: \ \mathbf{c_j} = q_j(u_j(\mathbf{y})), \mathbf{c_{j+1}} = q_{j+1}(u_{j+1}(\mathbf{y})), j \in \{2 \cdot i | i \in \mathbb{N}\}$$

The distance calculation can then be expressed by the following equation:

$$\hat{d}(\mathbf{r}, \mathbf{y})^2 = \sum_{j=1}^{m'} d(\langle u_{2j-1}(\mathbf{r}), u_{2j}(\mathbf{r}) \rangle, \langle \mathbf{c_{2j-1}}, \mathbf{c_{2j}} \rangle)^2 \quad (8)$$

To efficiently calculate this, the product quantization sequences consisting of m numbers can be transformed into sequences of $m' = \frac{m}{2}$ numbers. Therefore, all centroid id pairs $\langle id(\mathbf{c_j}), id(\mathbf{c_{j+1}}) \rangle$ can be transformed into single ids:

$$id(\mathbf{c_j}, \mathbf{c_{j+1}}) = id(\mathbf{c_j}) \cdot |C| + id(\mathbf{c_{j+1}}) \quad (9)$$

This is called the `SHORT_CODES` mode. For a kNN-Join, this transformation process has to be done only once irrespective of the number of queries (see Fig. 4b Line 3 to 6). Optimal settings for the threshold Th_{flex} are discussed in Sect. 5.4.

4.6 Optimizations

Target List for Product Quantization Search A naïve way of doing the distance calculation via product quantization might be to calculate the distances directly by iterating through the targets instead of collecting the targets as it is done in Fig. 4b in Line 13. However, to execute product quantization efficiently it is important that the precomputed distances stay in the cache. Since the precomputed distances are specific for the query, it is necessary to collect all product quantization sequences and assign them to the query vectors in the first place. Afterward, the distance computation can be done query-wise. So, all precomputed distances specific for a query can stay in the cache. Moreover, the approach of [AKLS15] could be used to further improve memory locality to speed up the product quantization search. Thereby, product quantization sequences are compressed to fit into SIMD cache lines.

Prefetching As stated in Sect. 4.4, we collect the targets in Line 13 of the search algorithm (Fig. 4b) and assign them to the query vectors they should be compared to. This requires a lot of random memory accesses to the lists of targets. To speed up this step, we prefetch the target list entries which has to be updated next from time to time. We tested the effect of the prefetching with our algorithm on a query with 10,000 queries and 100,000 targets (300-dimensional vectors) and $\alpha = 100$ and $k = 10$. For this query the construction time of the target list could be reduced by $\approx 35\%$, from 1.4 seconds to 0.9 seconds. For more

details about that one can take a look at the code[3].)
Fast Top-K Update In Line 19 of the algorithm in Fig. 4b the top_k gets updated after every distance calculation. If the distance value is lower than every other index entry, the new index entry has to be inserted into this array of current nearest neighbors. However, this can be time-consuming since every other array element with a larger distance has to be moved. For a large top_k, the updates can be accelerated by first adding new candidates in a buffer. If this buffer gets full or all distance calculations are done, all candidates are added to the top_k in one run. This is in particular useful if post verification (see Sect. 4.4) should be done and thus a large set of candidates is required in the first place. Alternatively, one can use a linked list instead of an array for the top_k. However, linked lists are space consuming which could become a problem for large query sets.

5 Evaluation

In this section, we first evaluate our adaptive kNN-Join implementation for varying post-verification factors and α values and compare them to the basic batch-wise product quantization approach (see Sect. 5.2). Moreover, we provide a detailed runtime investigation for the different sub-routines of the kNN-Join given different query and target set sizes (Sect. 5.3). The impact of short and long code sizes on the precomputation and distance calculation is shown in Sect. 5.4. In Sect. 5.5, we finally evaluate the accuracy of the target size estimator that was presented in Sect. 4.3.

5.1 Experimental Setup

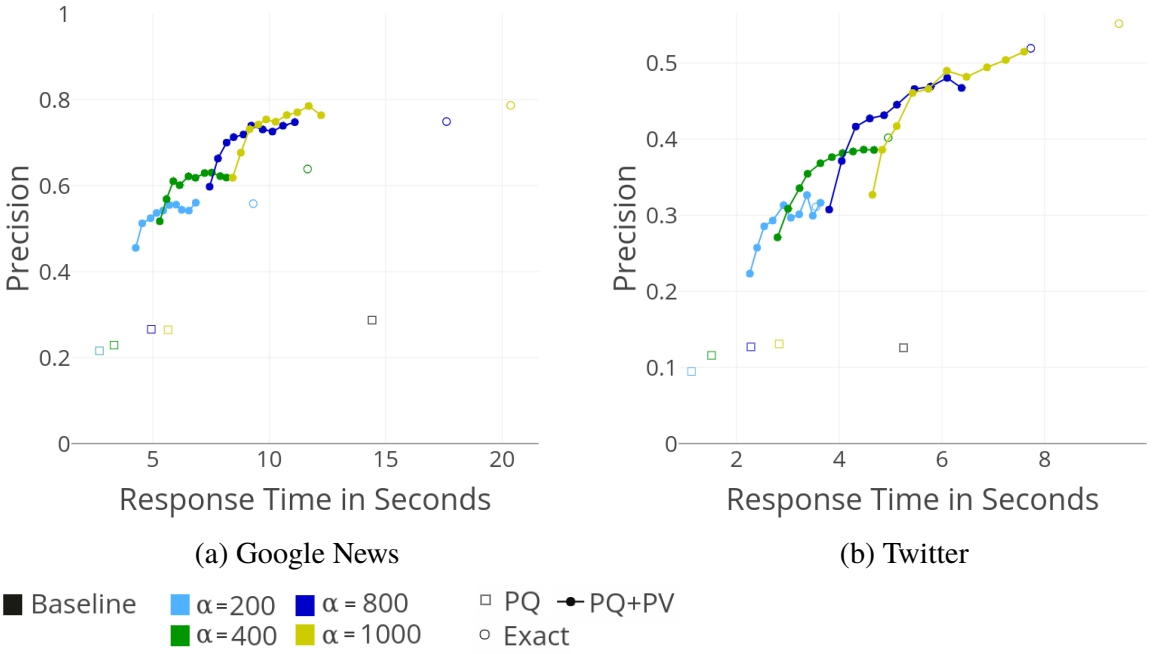
We use two different datasets of word embeddings to evaluate our approach, the popular Google News dataset[4] which is trained with the word2vec [Mi13] skip gram model and a dataset trained on data from Twitter[5] with GloVe [PSM14]. We use python scripts to create the index structures for these datasets as shown in Tab. 1. The kNN-Join, that can be used for queries similar to the example in Fig. 1, is implemented as a user-defined function.
The index consists of entries with an `entry_id` and a product quantization sequence as well as a codebook storing the centroids. As a baseline, we use the exhaustive product quantization search as described in [JDS11], which can easily generalized to a kNN-Join operation. Basically, it makes no use of inverted indexing and thus calculates approximated distance values between any query vector in R and any target vector in T to determine the kNN results. The method can simply reuse the index data table and the codebook of our adaptive index (Page 6 Fig. 3) while ignoring the `Coarse ID` column. To make the comparison fair we implemented a batch-wise search algorithm as UDF, like it is done for the

[3] `https://github.com/guenthermi/postgres-word2vec/blob/master/freddy_extension/ivpq_search_in.c`
[4] `https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing`, last access: 15.08.18
[5] `https://nlp.stanford.edu/projects/glove/`, last access: 15.08.18

	Google News (GN)	**Twitter (TW)**
Size	3, 000, 000	1, 193, 514
Dimensionality	300	100
Coarse Centroids	$2 \cdot 32$	$2 \cdot 20$
Product Quantization	$m = 30,\ \|C\|^* = 32$	$m = 10,\ \|C\|^* = 32$
Confidence	$P_{conf} = 0.8$	$P_{conf} = 0.8$
Threshold (For Flexible Product Quantization)	$Th_{flex} = 15,000$	$Th_{flex} = 15,000$

$*$ number of centroids for each quantizer

Tab. 1: Dataset and Index Characteristics



(a) Google News

(b) Twitter

Fig. 7: Evaluation of Execution Time and Precision

adaptive search algorithm. To enable repeatability we have published the implementation[6]. The machine we run the evaluation on is a Lenovo ThinkPad 480s with 24GB main memory, an Intel i5-8250U CPU (1.6GHz) and a 512GB SSD. The computation runs only on a single core in a PostgreSQL instance on a Ubuntu 16.04 Linux System.

5.2 Influence of Index Parameters on Precision and Execution Time

In Fig. 7, the execution time and precision curves for different α values and increasing *pvf* values are shown. All the kNN-Joins are executed on 5,000 query and 100,000 target vectors with $k = 5$. The post verification factors used for the computation are $10, 20, \ldots, 100$. The precision is determined by calculating the amount of nearest neighbor results of a query vector which concur with the exact results relative to the number of k. Since doing the exact

[6] `https://github.com/guenthermi/postgres-word2vec`

calculation for all query vectors of a kNN-Join is very time-consuming, we draw bootstrap samples of the query vectors of size 100 to derive an estimation of the actual precision value by determining the precision of the samples results. The measurements for every configuration are done 20 times and the median values are determined. The value of $\alpha \cdot k$ is always lower than Th_{flex}. Thus, the `LONG_CODES` method is used.
As one can see, for most of the chosen values of *pvf* and α the adaptive search with PQ distance calculation has the shortest execution time and also outperforms the product quantization baseline method in terms of precision and runtime. Join operations with exact distance calculation have significantly longer execution times than the other methods, however achieving the highest precision value. Nevertheless, the post verification might be the better choice in most of the cases, since it achieves high precision values while being much faster than the exact computation. For increasing values of *pvf* the execution time, but also the precision generally increases. Regarding the α values, one can also observe that higher values lead to higher precision values at the expense of execution time.
The post verification method is significantly slower than the product quantization method, even though *pvf* has a low value. This is the case since the use of post verification requires to at least calculate k exact distance values for each query vector. Furthermore, it needs to retrieve the raw vector data for every target vector which has to be considered for distance calculation. Moreover, it is necessary to hold these vectors in memory until the distance computation starts. During the distance computation, the vectors of the currently best candidates have to be stored together with the product quantization sequences in a separate TopK list to apply the post verification step later. For high values of *pvf*, on one hand, the post verification step gets time-consuming while on the other hand more updates of the TopK lists are required during the distance calculation step.

5.3 Performance Measurements

We evaluate the performance of the search algorithm by measuring the execution time of certain subroutines of the algorithm denoted by numbers 1 to 4 in Fig. 4a. This is done for different cardinalities of query sets R and target sets T. The query and target vectors are sampled from the whole set of word embeddings of the Google News dataset. The results of our measurements are shown in Fig. 8 for different values of $|R|$ and $|T|$. For the measurements we set $\alpha = 100$, $pvf = 10$ and used a fixed target set size of $10,000$ while increasing $|R|$ and a query set size of $10,000$ while increasing $|T|$. All measurements are done five times and the average value is determined. The distance computation time increases with the query set size as well as with the number of target vectors. The query construction time only increases with an increasing query set size. If the query set size is fixed the query construction time slightly decreases with increasing target set size because a higher number of partitions has to be determined for every query vector in case the number of targets is very low. The main effort during the query construction is the calculation of the coarse quantization for every query vector. Since this process does not change with the number of target vectors, the execution time is rather constant. The data retrieval time

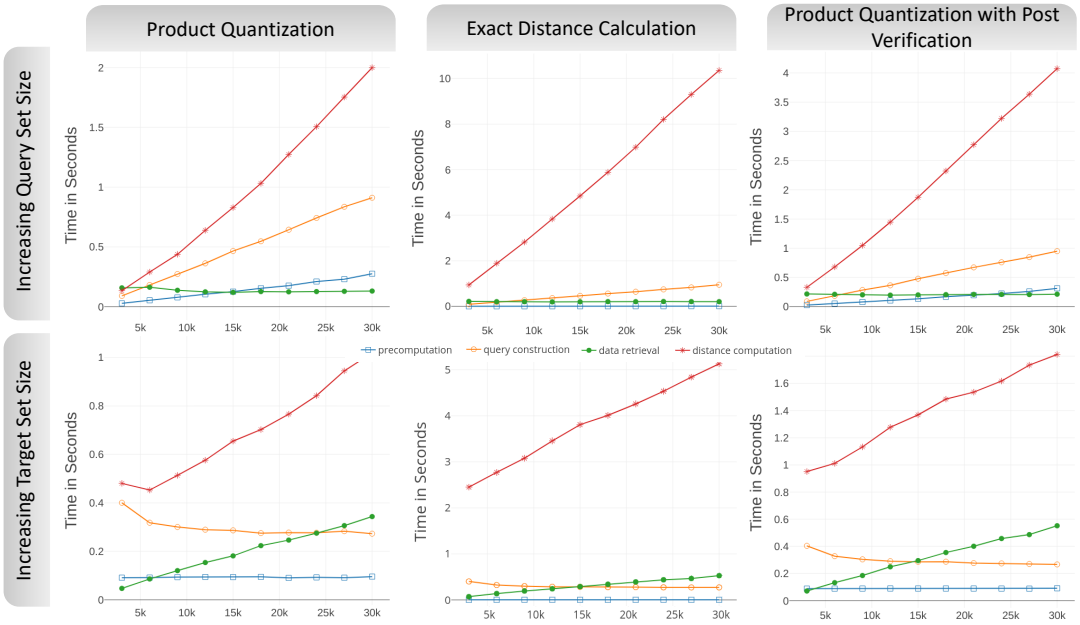


Fig. 8: Time Measurement for increasing sizes of query set R and target set T

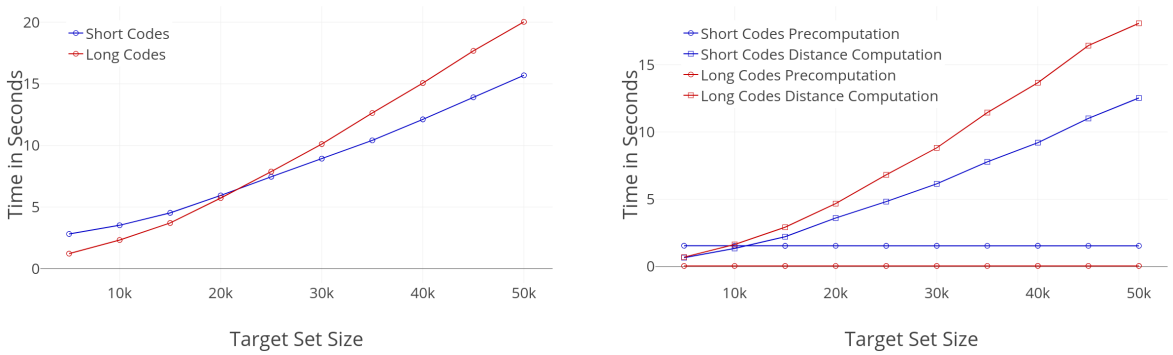
effort is nearly constant for an increasing number of query vectors while its execution time increases if the target vector set grows. The preprocessing has to be done per query vector. Therefore only the query set size influences its execution time.

5.4 Flexible Product Quantization

Flexible product quantization (Sect. 4.5) can adjust the product quantization distance calculation to smaller or bigger sets of vectors. The product quantization sequences in our index structure for the Google News dataset consist of codes $c_i \in \{0, \ldots, 31\}$ of length 30 which can be combined to shorter codes $c'_i \in \{0, \ldots, 1023\}$ with length 15. The overall execution time of a kNN-Join with product quantization distance calculation is shown for both methods in Fig. 9a. Fig. 9b shows the execution times for the precomputation and distance calculation step. We use query sets of size $5,000$. The target set size $|T|$ is shown on the x axis. The value α is set to $\frac{|T|}{2 \cdot k}$. The measurements are done 10 times with randomly sampled query and target vectors and average values are determined. For small target sizes with $|T| \leq 20,000$ the computation via long codes is faster. For larger target sets the overhead of the distance calculation for long codes becomes prevalent such that the calculation with short codes is faster.

5.5 Accuracy of the Target Size Estimation

The number of targets determined in the retrieval step of the algorithm before the distance calculation can be estimated. For this purpose we leverage an approximation of the



(a) Execution Times of Both Methods (b) Precomputation and Distance Computation

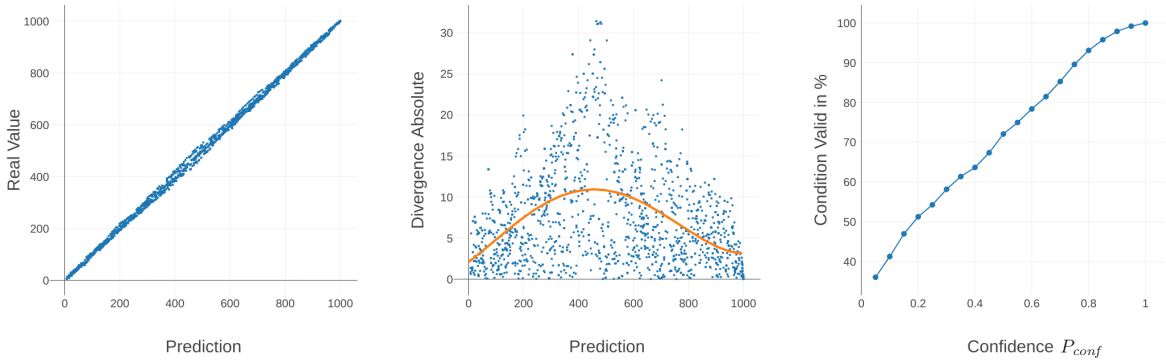
Fig. 9: Evaluation of Short and Long Codes Calculation

hypergeometric distribution as described in Sect. 4.3. The estimated number of targets derived from the index is μ as defined in Eq. (3). In Fig. 10a, a scatter plot of the estimated and actually derived number of targets is shown. For these measurements, kNN-Joins with a single randomly sampled query vector are executed and the number of targets obtained in the first retrieval step is determined inside the user-defined function. This was done for all $\alpha \in \{1, \ldots, 100\}$, $k = 5$ and target sets of size $|T| = 1,000$. For each α value 10 queries are executed. The divergence of the estimation is higher if the desired number of targets per query vector gets higher. This can be noticed in the 4th-grade polynomial regression curve of the sample points in Fig. 10b. However, if the number of desired targets is near to $|T|$ it is apparently decreasing.

To prevent the system from executing a lot of database queries, one can adjust the confidence value P_{conf}. It represents how likely it is that only one database request is sufficient to derive the desired number of targets from the index. This was also evaluated by single query kNN-Joins with $\alpha = 10$ and the same search parameters as in the last experiment. The amount of queries where the condition is true (only one request was required) in relation to P_{conf} is shown in Fig. 10c. For each confidence value $P_{conf} \in \{0.05 \cdot i | i = 1, \ldots, 20\}$ 1,000 queries are executed. As desired, the amount of queries where the condition is true rises up to 100%, if the confidence value increases up to 1. However, the actual confidence is quite higher than P_{conf}, since the confidence can only be increased step-wise by incrementing ω.

6 Related Work

There is already limited work done in integrating kNN operations in database systems. For instance, PostgreSQL can be extended by PostGIS [Po18] which allows running kNN queries for low dimensional (geographical) data. Index structures can be created with GiST (Generalized Search Trees) to speed up such operations. An integration of vector similarity search for high dimensional data into Spark has been done by [BBS17] for word embeddings. Here, LSH [Ch02] or spherical k-means [DM01] is used to partition



(a) Estimated Size Values (b) Divergence (Absolute) (c) Confidence

Fig. 10: Estimation of Target Set Size

vectors for filtering. However, the actual distance calculation might be done with exact methods. We proposed FREDDY [Gü18] which supports approximated kNN queries for high dimensional data, however, can not efficiently execute kNN-Join operations. A system called ADAM$_{pro}$ [GAKS14] adds approximated kNN-Search techniques on top of a database system for multimedia retrieval. Furthermore, also approximated kNN-Joins are already integrated into a relational database system by [YLK10]. However, this is only applicable for low dimensional data. This work differs from previous work in the way that it employs state-of-the-art approximated nearest neighbor search techniques to support approximated kNN-Joins also for high dimensional data. To do so several modifications and optimization specific for kNN-Joins in RDBMSs have been done.

Approximated Nearest Neighbor Search The difficulty to find the nearest neighbors especially in high dimensional vector spaces has led to the development of several kinds of approaches for approximated nearest neighbor (ANN) search. However, not all of them are applicable for kNN-Join operations in RDBMSs. On one hand, recently graph-based methods are developed [Ha11] which are fast, however, do not allow online index updates. On the other hand, there are so-called cell probe methods which divide the search space into several cells. One of the most popular ones E2LSH [Da04] applies locality sensitive hashes (LSH) to achieve such a partitioning. Jegou et al. [JDS11] employ product quantization for partitioning. Additionally, their approach can be combined with inverted indexing similar to [SZ03] for even faster search. One advantage of product quantization is, that it is easy to add vectors during runtime in an online update manner which is particularly useful for the application in relational database systems. Thus, we based our index on such quantization techniques. In the context of word embeddings, vectors for text values of multiple tokens can be added during runtime by an averaging method [CJ15]. For such updates, the quantizations of the new vectors have to be calculated. Vectors can be removed by simply deleting the quantization entries from the index. For frequent inserts and deletions, the online product quantization approach proposed by [XTZ18] suggests

updating the centroids of the quantizer functions. In this way, the index can react to context drifts of its containing data.

7 Conclusion

We propose a novel index structure for approximated kNN-Joins on high dimensional data which is flexible enough to deliver good performance on different cardinalities for query and target vector set. It enables non-exhaustive search for different target sets. We have shown that the proposed index structure can achieve faster response times than product quantization as an instance of a state-of-the-art exhaustive search method. We provide a practical implementation of the operator and our optimizations in PostgreSQL.

Future Work By adjusting the post verification factor *pvf* one can influence the precision and response time of the kNN-Join. However, there are further search parameters, in particular α, which have an influence here. It might be hard for a user to find out which parameter configuration leads to a specific precision and response time. Thus, an oracle which can provide the optimal parameters to achieve a good precision by fulfilling constraints regarding the execution time would improve the usability of the system.

Acknowledgments

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907) and by the Intel® AI Research.

References

[AKLS15] André, Fabien; Kermarrec, Anne-Marie; Le Scouarnec, Nicolas: Cache Locality is Not Enough: High-performance Nearest Neighbor Search with Product Quantization Fast Scan. Proc. of the VLDB Endowment, 9(4):288–299, 2015.

[BBS17] Bordawekar, Rajesh; Bandyopadhyay, Bortik; Shmueli, Oded: Cognitive Database: A Step towards Endowing Relational Databases with Artificial Intelligence Capabilities. CoRR, abs/1712.07199, 2017.

[Be99] Beyer, Kevin; Goldstein, Jonathan; Ramakrishnan, Raghu; Shaft, Uri: When Is "Nearest Neighbor" Meaningful? In: Proc. of the 7th International Conference on Database Theory. Springer, pp. 217–235, 1999.

[BL12] Babenko, Artem; Lempitsky, Victor: The Inverted Multi-Index. In: 2012 IEEE Conference on Computer Vision and Pattern Recognition. IEEE, pp. 3069–3076, 2012.

[Ch02] Charikar, Moses S: Similarity Estimation Techniques from Rounding Algorithms. In: Proc. of the 34th Annual ACM Symposium on Theory of Computing. ACM, pp. 380–388, 2002.

[CJ15] Campr, Michal; Ježek, Karel: Comparing Semantic Models for Evaluating Automatic Document Summarization. In: International Conference on Text, Speech, and Dialogue. Springer, pp. 252–260, 2015.

[Da04] Datar, Mayur; Immorlica, Nicole; Indyk, Piotr; Mirrokni, Vahab S: Locality-Sensitive Hashing Scheme Based on p-stable Distributions. In: Proc. of the 20th Annual Symposium on Computational Geometry. ACM, pp. 253–262, 2004.

[DM01] Dhillon, Inderjit S; Modha, Dharmendra S: Concept Decompositions for Large Sparse Text Data Using Clustering. Machine Learning, 42(1-2):143–175, 2001.

[GAKS14] Giangreco, Ivan; Al Kabary, Ihab; Schuldt, Heiko: ADAM - A Database and Information Retrieval System for Big Multimedia Collections. In: 2014 IEEE International Congress on Big Data. IEEE, pp. 406–413, 2014.

[Gr84] Gray, Robert: Vector Quantization. IEEE ASSP Magazine, 1(2):4–29, 1984.

[Gü18] Günther, Michael: FREDDY: Fast Word Embeddings in Database Systems. In: Proc. of the 2018 International Conference on Management of Data. ACM, pp. 1817–1819, 2018.

[Ha11] Hajebi, Kiana; Abbasi-Yadkori, Yasin; Shahbazi, Hossein; Zhang, Hong: Fast Approximate Nearest-Neighbor Search with k-Nearest Neighbor Graph. In: Proc. of the 22nd International Joint Conference on Artificial Intelligence. volume 22, p. 1312, 2011.

[JDS11] Jegou, Herve; Douze, Matthijs; Schmid, Cordelia: Product Quantization for Nearest Neighbor Search. IEEE Transactions on Pattern Analysis and Machine Intelligence, 33(1):117–128, 2011.

[LG14] Levy, Omer; Goldberg, Yoav: Linguistic Regularities in Sparse and Explicit Word Representations. In: Proc. of the 18th Conference on Computational Natural Language Learning. pp. 171–180, 2014.

[Mi13] Mikolov, Tomas; Sutskever, Ilya; Chen, Kai; Corrado, Greg S; Dean, Jeff: Distributed Representations of Words and Phrases and their Compositionality. In: Advances in Neural Information Processing Systems. pp. 3111–3119, 2013.

[Po18] PostGIS, postgis.net, Last Access: 06.09.2018.

[PSM14] Pennington, Jeffrey; Socher, Richard; Manning, Christopher D.: GloVe: Global Vectors for Word Representation. In: Empirical Methods in Natural Language Processing (EMNLP). pp. 1532–1543, 2014.

[SZ03] Sivic, Josef; Zisserman, Andrew: Video Google: A Text Retrieval Approach to Object Matching in Videos. In: Proc. of the 9th IEEE International Conference on Computer Vision-Volume 2. IEEE Computer Society, p. 1470, 2003.

[XTZ18] Xu, Donna; Tsang, Ivor; Zhang, Ying: Online Product Quantization. IEEE Transactions on Knowledge and Data Engineering, 2018.

[YLK10] Yao, Bin; Li, Feifei; Kumar, Piyush: K Nearest Neighbor Queries and kNN-Joins in Large Relational Databases (Almost) for Free. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). IEEE, pp. 4–15, 2010.