Partial Refinement for Similarity Search with Multiple Features

Marcel Zierenberg

Brandenburg University of Technology Cottbus - Senftenberg
Institute of Computer Science, Information and Media Technology
Chair of Database and Information Systems
P.O. Box 10 13 44, 03013 Cottbus, Germany
zieremar@tu-cottbus.de

Abstract. Filter refinement is an efficient and flexible indexing approach to similarity search with multiple features. However, the conventional refinement phase has one major drawback: when an object is refined, the partial distances to the query object are computed for *all features*. This frequently leads to more distance computations being executed than necessary to exclude an object. To address this problem, we introduce *partial refinement*, a simple, yet efficient improvement of the filter refinement approach. It incrementally replaces partial distance bounds with exact partial distances and updates the aggregated bounds accordingly each time. This enables us to exclude many objects before all of their partial distances have been computed exactly. Our experimental evaluation illustrates that partial refinement significantly reduces the number of required distance computations and the overall search time in comparison to conventional refinement and other state-of-the-art techniques.

1 Introduction

Similarity search with multiple features is an effective way of finding objects similar to a query object. Instead of using only a single *feature* for the comparison of objects (e.g., a single color histogram for the comparison of images), multiple features (e.g., color, edge and texture features) are utilized. A *distance function* assigned to each feature is employed to compute the respective *partial distances* (dissimilarities) between each of the compared objects' features. These partial distances are combined into an *aggregated distance* by means of an *aggregation function*. Finally, the most similar objects are determined according to the lowest aggregated distances to the query object.

Indexing approaches to similarity search [1, 2] aim to exclude as many objects as possible from the search to decrease CPU and I/O costs for the computation of distances.

Filter refinement is a well-known technique and utilized by several indexing approaches to multi-feature similarity search (e.g., [3, 4, 5, 6]). In general, the *filtering phase* aims to discard objects based on inexpensively computed approximations of the distance between the query and each database object (*bounds*). The *refinement phase* then computes the exact distances for the remaining candidates to determine the most similar objects. For search with multiple features, *partial bounds* for each feature are combined into an *aggregated bound*. The exclusion of objects in the filtering and refinement phase is based on those aggregated bounds.

Unfortunately, the performance of filter refinement deteriorates with an increasing number of features. As the *(intrinsic) dimensionality* [7] of the aggregation function rises, the *approximation error* of the aggregated bounds increases as well. A higher approximation error results in less efficient search because fewer objects can be excluded.

1.1 Contribution

The main contribution of this paper is the improvement of the refinement step for filter refinement with multiple features. *Conventional refinement* (see Section 3) manages objects with the help of a candidate list sorted in ascending order according to the aggregated lower bounds. When an object on top of the candidate list is refined, all of the object's partial distances are computed exactly and combined into an aggregated distance. Unfortunately, this frequently leads to more partial distances being computed than necessary to exclude objects.

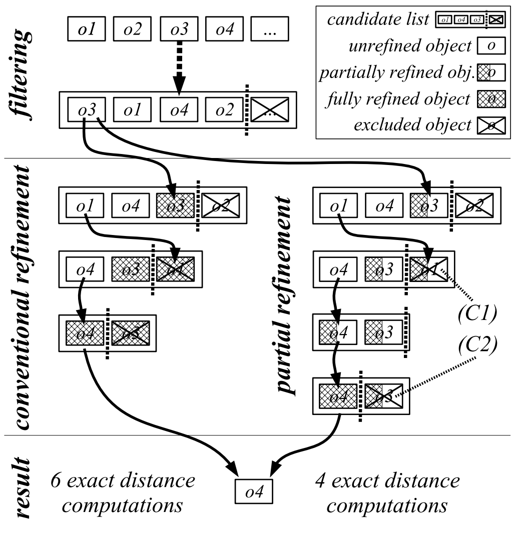
In contrast, our *partial refinement* approach (see Section 5) incrementally replaces partial distance bounds of objects with their exact partial distances, updates the aggregated bounds and reinserts the objects into the candidate list. This allows us to gradually tighten the aggregated bounds and to exclude many objects before all of their partial distances have been computed exactly.

Example 1. Consider for example a similarity search with two features as depicted in Figure 1a. The filtering phase produces a candidate list ordered according to the aggregated lower bounds. Conventional refinement requires three iterations and each iteration executes two distance computations (*fully refined*).

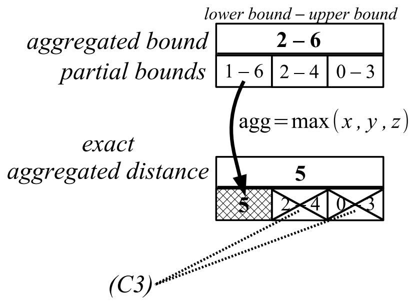
In contrast, partial refinement computes only one partial distance per iteration (*partially refined*), updates the aggregated bounds of the object and reinserts it into the candidate list. In this case, it permits a *direct* (case C1) and a *delayed exclusion* (case C2) of two objects, without computing all of their exact partial distances. While the conventional refinement approach requires six distance computations in this example, four distance computations are sufficient to determine the most similar object with partial refinement.

Example 2. Another example for the benefits of partial refinement is the *partial exclusion* of objects (case C3) for specific aggregation functions like the minimum or maximum function. If it becomes obvious that a specific partial distance does not influence the aggregated distance, it can be safely excluded from computation. For the example of the maximum function with three features in Figure 1b, two partial distance computations can be excluded because their upper bound (4 and 3) is lower than the exact partial distance of the first feature (5).

To demonstrate the efficiency of our approach, we experimentally compare partial refinement to the linear scan, conventional filter refinement [6], the *Onion-tree* [8] and the



(a) Conventional vs. partial refinement (two features)



(b) Partial refinement for the maximum function (three features)

Fig. 1. Examples for direct (C1), delayed (C2) and partial exclusion (C3).

Threshold Combiner Algorithm [9] (see Section 6). The evaluation illustrates that partial refinement is able to significantly reduce the number of required distance computations and the overall search time.

2 Preliminaries

This section defines the notations and terms used throughout this paper.

2.1 Nearest Neighbor Search

Similarity search can be performed by means of a *k-Nearest Neighbor query*. A $k\text{NN}(q)$-query in the universe of objects $\mathbb{U}$ returns k objects out of a database $DB = \{o^1, \ldots, o^n\} \subseteq \mathbb{U}$ that are closest (most similar) to the query object $q \in \mathbb{U}$. The distance between objects is computed by a distance function $\delta : \mathbb{U} \times \mathbb{U} \mapsto \mathbb{R}_{\geq 0}$ that operates on the features $\hat{q}$ and $\hat{o}^i$ extracted from the objects. The result is a (non-deterministic) set K with $|K| = k$ and $\forall o^i \in K, o^j \in DB \setminus K : \delta(q, o^i) \leq \delta(q, o^j)$.

A *multi-feature kNN-query* substitutes the single features $\hat{q}$ and $\hat{o}^i$ with m features $\hat{q} = (\hat{q}_1, \ldots, \hat{q}_m)$ and $\hat{o}^i = (\hat{o}^i_1, \ldots, \hat{o}^i_m)$. A distance function δ_j is assigned to each single feature to compute the *partial distances* $d^i_j = \delta_j(q, o^i)$. An *aggregation function* $\text{agg} : \mathbb{R}^m_{\geq 0} \mapsto \mathbb{R}_{\geq 0}$ combines all partial distances to an *aggregated distance* d^i_{agg} and the k nearest neighbors are then determined according to the aggregated distance.

An optional *weighting scheme* with weights $W = (w_1, \ldots, w_m)$ and $\forall w_j \in W : w_j \geq 0$ can be applied to the features of the aggregation function. These weights are typically unknown at the time of index construction. Instead, they are dynamically determined at query time in order to optimally adapt the aggregation function to the query object and the demands of the user [10].

2.2 Metric Indexing

A *metric* is a distance function with the properties *positivity* ($\forall x \neq y \in \mathbb{U} : \delta(x, y) > 0$), *symmetry* ($\forall x, y \in \mathbb{U} : \delta(x, y) = \delta(y, x)$), *reflexivity* ($\forall x \in \mathbb{U} : \delta(x, x) = 0$) and *triangle inequality* ($\forall x, y, z \in \mathbb{U} : \delta(x, z) \leq \delta(x, y) + \delta(y, z)$).

Metric indexing approaches [1, 2] exclude objects from search by computing *bounds* of the distance from the query object to database objects. The lower bound lb_j^i and upper bound ub_j^i of the exact partial distance $d_j^i = \delta_j(q, o)$ between query object q and database object o^i can be determined by exploiting the triangle inequality and the precomputed distance to a *reference object* (*pivot*) p as follows:

$$lb_j^i = |\delta_j(q, p) - \delta_j(p, o^i)| \leq \delta_j(q, o^i) \leq \delta_j(q, p) + \delta_j(p, o^i) = ub_j^i \,. \tag{1}$$

The *approximation error* $\epsilon_1^i, \ldots, \epsilon_m^i$ of the partial distance bounds is calculated by the weighted difference between the respective upper and lower bounds $\epsilon_j^i = w_j * (ub_j^i - lb_j^i)$.

The *intrinsic dimensionality* ρ is defined as $\rho = \frac{\mu^2}{2\sigma^2}$ where μ is the mean and σ^2 the variance of a distance distribution. It is frequently used as an estimator for the indexability of metric spaces [7].

2.3 Monotonicity and Aggregated Bounds

An aggregation function agg is *monotone increasing in the j-th argument* with $1 \leq j \leq m$, $d = (d_1, \ldots, d_j, \ldots, d_m)$ and $d' = (d_1, \ldots, d_j', \ldots, d_m)$ iff:

$$\forall d, d' \in \mathbb{R}^m_{\geq 0} : d_j < d_j' \implies \mathrm{agg}(d) \leq \mathrm{agg}(d') \,. \tag{2}$$

This means, if all arguments except d_j are constant and d_j is increased to d_j', the result of the aggregation function will either be constant or also increase.

An aggregation function agg is *globally monotone increasing* iff it contains only monotone increasing arguments. An example for a globally monotone increasing function is $\mathrm{agg}(d_1, d_2) = d_1 + d_2$.

For the sake of simplicity, we consider only globally monotone increasing aggregation functions for distances (dissimilarity values) in the following. However, note that the stated results are easily adaptable to other types, like locally or flexible monotone aggregation functions and aggregation functions for similarity values [6].

Even though the aggregation function can be a metric if all partial distance functions are also metrics (e.g., arithmetic mean or maximum of L_1 distances), this is not necessarily the case. The minimum and the median function ($m > 2$) are examples for non-metric aggregation functions that are globally monotone increasing.

The aggregated lower (upper) bound lb_{agg}^i (ub_{agg}^i) on the exact aggregated distance $d_{agg}^i = \mathrm{agg}\left(d_1^i, \ldots, d_m^i\right)$ of a globally monotone increasing aggregation function can be computed by inserting partial lower (upper) bounds for all features into the aggregation function:

$$lb^i_{agg} = \text{agg}\left(lb^i_1, \ldots, lb^i_m\right) \leq \text{agg}\left(d^i_1, \ldots, d^i_m\right) \leq \text{agg}\left(ub^i_1, \ldots, ub^i_m\right) = ub^i_{agg} \, . \tag{3}$$

The approximation error of the aggregated bounds is defined as $\epsilon^i_{agg} = ub^i_{agg} - lb^i_{agg}$.

3 Filter Refinement

The following section briefly summarizes the conventional filter refinement approach to similarity search with multiple features.

To build the index, metric filter refinement approaches [4, 6] compute one matrix of distances between pivots and database objects per feature. Algorithm 1 depicts the filtering phase for a kNN-query with multiple features. At first, bounds for each partial distance are computed based on the precomputed distance matrices and Equation (1) (line 3). Subsequently, these partial bounds are combined into aggregated bounds by Equation (3) (line 4). Objects having a higher aggregated lower bound lb^i_{agg} than the k-th lowest aggregated upper bound ub^i_{agg} seen so far (t_{max}) are excluded from the search (lines 5 and 9). The remaining objects are managed by a candidate list sorted in ascending order according to lb^i_{agg} (priority queue).

In the conventional refinement phase (Algorithm 2) the previously determined candidate objects have to be refined. Starting with the candidate with the lowest aggregated lower bound lb^i_{agg}, we check if the object appeared at the top of the candidate list before (line 3). If not, the object was not refined yet and the exact aggregated distance d^i_{agg} has to be computed (line 5). Afterwards, the object is either excluded because its exact aggregated distance is larger than the current threshold value t_{max} (line 6) or it is reinserted into the candidate list.

If the object at the top of the candidate list was already refined before (line 3), the object is one of the k nearest neighbors because the object's exact aggregated distance d^i_{agg} is lower than the remaining objects' aggregated lower bounds lb^i_{agg}. Refinement is stopped as soon as k nearest neighbors were found.

4 Related Work

This section gives an insight into the state-of-the-art of indexing for similarity search with multiple features and filter refinement for multiple features in particular.

If the aggregation function is a metric, an arbitrary (single-feature) metric index (e.g., *Onion-tree* [8]) can be build directly on top of the aggregated distances (*naïve approach*). Unfortunately, this solution prevents partial refinement and is inflexible because it requires the index to be rebuilt when the used aggregation function, features or weights are changed [6]. *Multi-metric indexing* [11] solves this problem partially. It defines a framework to transform arbitrary metric indexing approaches for single features into metric indices for multiple features with dynamic weighting. However, the restriction to metric aggregation functions remains.

Algorithm 1: Multi-feature kNN-query – filtering

```
Input: $k$, $q$, $DB$, $agg$, $W$
1  $t_{max} = \infty$;
2  foreach $o^i \in DB$ do
3      Compute partial bounds $lb^i_j$ and $ub^i_j$ for each feature;      // Equation (1)
4      Compute aggregated bounds $lb^i_{agg}$ and $ub^i_{agg}$;              // Equation (3)
5      if $lb^i_{agg} > t_{max}$ then continue;                             // exclude object?
6      else
7          candidates.insert($o^i$);
8          $t_{max} = k$-th lowest $ub^i_{agg}$;                             // update threshold
9          candidates.cut($t_{max}$);           // exclude objects with $lb^i_{agg} > t_{max}$
10 return candidates;
```

Algorithm 2: Multi-feature kNN-query – conventional refinement

```
Input: $k$, $q$, $candidates$, $t_{max}$, $agg$, $W$
1  repeat
2      $o^i$ = candidates.pop();                  // get candidate with lowest $lb^i_{agg}$
3      if $o^i$ is refined then results.insert($o^i$);              // already refined?
4      else
5          Compute exact aggregated distance $d^i_{agg}$;                 // refinement
6          if $d^i_{agg} > t_{max}$ then continue;                       // exclude object?
7          else ...;                               // (lines 7 - 9 of Algorithm 1);
8  until results.size() = $k$;
9  return results;
```

The *M*^{2}*-tree* [12] is a multi-dimensional extension of the well-known *M-tree*. It supports dynamic weighting as well as metric and non-metric aggregation functions. However, it is not suitable for partial refinement and has the disadvantage that its clustering may be inefficient if only a subset of all indexed features is used for a query.

An index comprised of one matrix of distances to pivot objects per feature is described in [4]. This allows efficient queries with subsets of the indexed features and dynamic weighting since each matrix can be accessed individually. Filter refinement is used to exclude objects. However, the approach does not utilize a candidate list to determine the order of objects and objects that were not excluded are always fully refined.

Our previous research introduced *FlexiDex* [6], a flexible metric index for (logic-based) multi-feature similarity search. The index has to be created only once but can be efficiently used for different types of aggregation functions, numbers of features and weighting schemes. Originally, FlexiDex fully refines each object. However, in the course of our research we adapted it to incorporate all concepts of partial refinement.

Combiner algorithms (e.g., *Threshold Algorithm* (TA) [9]) merge the result lists of subqueries for each single feature into an aggregated result list. Once an object is seen in one of the lists, missing partial distances are computed by random access. This behavior resembles conventional refinement. However, filter refinement uses a single candidate list and sorts it based on the aggregated bounds. This allows it to adapt better to the aggregation function than combiner algorithms.

Our research focuses on metric indexing since it is more flexible and suffers less from the *curse of dimensionality* [2] than *spatial indexing* [1]. Nonetheless, partial refinement can be easily adapted to improve spatial indices that rely on filter refinement for multi-feature search (e.g., *GeVAS* [3] or *ASAP* [5]).

5 Partial Refinement

This section presents our main contribution, the *partial refinement approach*, which deals with the major drawback of conventional refinement to compute all partial distances of an object at once. We describe our concept in detail and give a pseudo-code implementation of the approach.

5.1 Exclusion of Objects

The main idea behind the concept of partial refinement is to exclude objects before all of their partial distances have been computed by gradually improving the quality of their aggregated bounds. This is accomplished by updating the aggregated bounds each time a partial distance of an object is computed exactly.

The order of partial distance computations for each individual object is determined in the filtering phase. The partial distances with the highest approximation error ϵ^i_j are computed first in order to quickly reduce the aggregated approximation error ϵ^i_{agg}. The following cases C1 – C3 are considered after every update of the aggregated bounds.

Direct exclusion (C1). If the updated aggregated lower bound lb^i_{agg} has increased above the current search threshold t_{max}, the object can be directly excluded from search without exactly computing the remaining partial distances.

Delayed exclusion (C2). If the updated aggregated lower bound lb^i_{agg} has not increased above the search threshold t_{max}, the object can currently not be excluded. The object is then reinserted into the candidate list and its position in the list is redetermined according to the updated aggregated lower bound. Now, if the search threshold t_{max} decreases below the updated aggregated lower bound lb^i_{agg} before the object reappears at the top of the candidate list, it can be excluded without exactly computing its remaining partial distances.

Partial exclusion (C3). For specific aggregation functions (e.g., minimum or maximum function) partial distance computations of an object can be excluded as soon as it becomes obvious that they do not influence the exact aggregated distance (*dominated distances*). This is achieved by comparing all partial distance bounds lb^i_j and ub^i_j of an object among each other.

5.2 Updating Aggregated Bounds

Partial refinement relies on the assumption that the computation of aggregated bounds and reinsertion into the candidate list is inexpensive in comparison to the computation of a partial distance.

In contrast to conventional refinement, which only needs to store the aggregated bounds for each object at query time, partial refinement additionally requires $2m$ partial bound values (m lower and m upper bounds) per object. Furthermore, a bit array b^i consisting of m bits per object is needed. Initially set to `false`, a bit b^i_j is set to `true` after the partial distance d^i_j for object o^i has been computed exactly.

It can easily be shown that replacing partial bounds with exact partial distances in Equation (3) can only result in tighter aggregated bounds. With each newly computed partial distance, the approximation error of the aggregated distance bounds ϵ^i_{agg} can be reduced. The *updated aggregated lower bounds* $\widehat{lb}^i_{agg}$ replace the old bounds after their computation and are defined as follows:

$$\widehat{lb}^i_j = \begin{cases} d^i_j, \text{ if } b^i_j = \texttt{true} \\ lb^i_j, \text{ otherwise} \end{cases} , \tag{4}$$

$$\widehat{lb}^i_{agg} = \operatorname{agg}\left(\widehat{lb}^i_1, \ldots, \widehat{lb}^i_m\right) \geq lb^i_{agg} \,. \tag{5}$$

The *updated aggregated upper bounds* $\widehat{ub}^i_{agg}$ are defined analogously.

Obviously, after all partial distances of object o^i have been computed, the updated aggregated distance bounds are equal to the exact aggregated distance d^i_{agg}:

$$(b^i_1 \wedge \ldots \wedge b^i_m) = \texttt{true} \implies \widehat{lb}^i_{agg} = d^i_{agg} = \widehat{ub}^i_{agg} \,. \tag{6}$$

5.3 Dominated Distances

Depending on the aggregation function (e.g., minimum or maximum function), it is not always necessary to compute all partial distances to determine the exact aggregated distance (case C3). In the following we will refer to those partial distances that are not needed as *dominated distances*.

For the example of the maximum function $\operatorname{agg}_{\max}$, a partial distance d^i_j is dominated if a partial lower bound lb^i_x exists that is greater or equal to the partial upper bound ub^i_j:

$$\begin{aligned} &\exists x \in \{1, \ldots, m\} : x \neq j \wedge lb^i_x \geq ub^i_j \implies \\ &\qquad \operatorname{agg}_{\max}\left(d^i_1, \ldots, d^i_j, \ldots, d^i_m\right) = \operatorname{agg}_{\max}\left(d^i_1, \ldots, d^i_{j-1}, d^i_{j+1}, \ldots, d^i_m\right) \,. \end{aligned} \tag{7}$$

This means the partial distance d^i_j does not influence the aggregation result (maximum) as it cannot be the largest distance. We can therefore safely exclude the partial distance from computation. In this case, bit b^i_j is set to `true` and d^i_j is set to the partial upper bound ub^i_j.

5.4 Partial Refinement Algorithm

Finally, we present the pseudo-code of partial refinement (see Algorithm 3). The conventional refinement of Algorithm 2 is adapted to incorporate the concepts presented in sections 5.1 – 5.3: the computation of a single partial distance (line 5), the detection of

Algorithm 3: Multi-feature kNN-query – partial refinement

```
Input: $k$, $q$, $candidates$, $t_{max}$, $agg$, $W$
repeat
    $o^i$ = $candidates$.pop();                       // get candidate with lowest $lb^i_{agg}$
    if $(b^i_1 \wedge \ldots \wedge b^i_m) = $ true then $results$.insert($o^i$);   // Equation (6)
    else
        Compute next exact partial distance $d^i_j$ and set $b^i_j = $ true;   // partial ref.
        Check for dominated distances and update $b^i$ accordingly;   // Equation (7)
        Compute updated aggregated bounds $\widehat{lb}^i_{agg}$ and $\widehat{ub}^i_{agg}$;   // Equation (5)
        if $\widehat{lb}^i_{agg} > t_{max}$ then continue;            // exclude object?
        else $\ldots$;                     // (lines 7 - 9 of Algorithm 1) ;
until $results$.size() $= k$;
return $results$;
```

dominated distances (line 6), the update of the aggregated bounds (line 7) and the check for the object's exclusion or reinsertion into the candidate list, based on the updated aggregated bounds (lines 8 and 9).

Depending on the memory constraints of the system, disk-based or in-memory indexing can be utilized. A disk-based implementation of the filter refinement approach to multi-feature similarity search is described in [6] and also applicable to partial refinement. There, each distance matrix is compressed and stored in the form of a compact *signature file* that can be sequentially read from disk. For in-memory indexing all needed distance matrices are simply preloaded into main memory.

6 Evaluation

This section presents the experimental evaluation. We demonstrate that partial refinement can vastly reduce the number of required distance computations and the overall search time in comparison to conventional refinement and other state-of-the-art approaches.

6.1 Experimental Setup

Partial refinement was compared to the linear scan, conventional refinement [6], an Onion-tree [8] build on top of aggregated distances and the Threshold Combiner Algorithm [9] based on m (single-feature) Onion-trees in connection with the *HS-Algorithm* [13]. As recommended by the authors, all Onion-trees were built with the keep-small strategy [8].

All experiments were run on a 2×2.26 GHz Quad-Core Intel Xeon with 8 GB RAM and an HDD with 7,200 rpm. However, we restricted our experiments to a single CPU core since the provided implementation of the Onion-tree does not support parallelization.

We utilized the image collections *Caltech-256 Object Category Dataset* [14] (30,607 images) and *ImageCLEF WEBUPV Image Annotation Dataset* [15] (250,000 images) for our experiments. Efficiency was assessed by measuring the average number of distance

Table 1. Used features, distance functions δ and intrinsic dimensionality ρ.

Feature	δ	**ρ per collection**	
		Caltech256	**WEBUPV**
CEDD	L_2	12.05	11.70
FCTH	L_1	5.77	6.35
EdgeHistogram	weighted L_1	8.55	9.97
DominantColor	EMD + L_2	2.16	1.91
ColorHistogram	dynamic QFD	11.48	10.47

computations and the average search time (wall-clock time) of kNN-queries for 100 randomly chosen query objects.

Features of varying intrinsic dimensionality and distance computation cost were chosen to examine the performance in distinct scenarios. Table 1 summarizes the used features and distance functions δ (Minkowski (L_p), Earth Mover's (EMD) and Quadratic Form (QFD)) and depicts the according intrinsic dimensionality ρ.

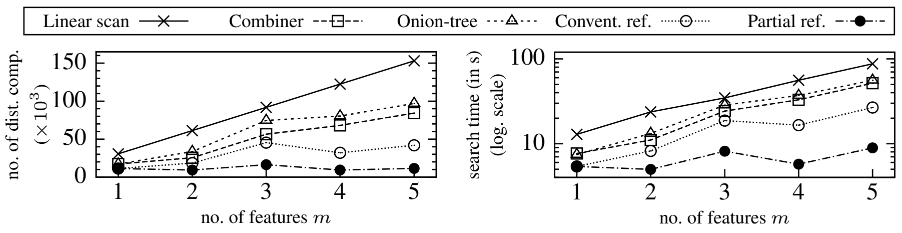
We used 64 pivot objects (randomly selected) per feature for filter refinement and kept all index data in main memory. Per feature, each object occupied 512 bytes of memory for the distances to the pivot objects (64×8 bytes; double precision), 16 bytes for the lower and upper partial distance bounds and 1 bit for the boolean flag b^i_j. Additional 16 bytes per object were required for the aggregated lower and upper bounds.

6.2 Aggregation Functions and Number of Features

The performance of partial refinement was investigated for various aggregation functions and numbers of features m. The features were added in the same order as given in Table 1 (from top to bottom).

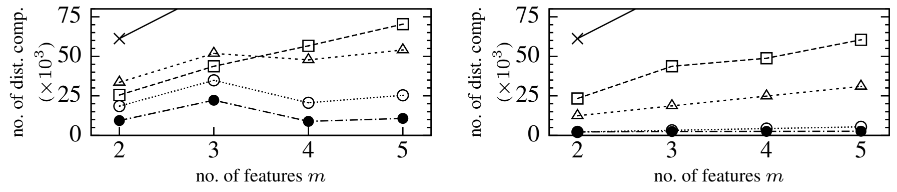
Figures 2a and 2b show the number of required distance computations and search time for $10-$NN-queries with the arithmetic mean. Obviously, the results of conventional and partial refinement were the same for a single feature. However, with an increasing number of features, partial refinement considerably outperformed all other approaches. It required up to 70 % less distance computations and up to 63 % less search time than conventional refinement. This means that the overhead of partial refinement (recomputing aggregated bounds and reinserting objects into the candidate list) is rather low in comparison to the time saved trough the reduced number of distance computations.

The number of required distance computations for 10-NN-queries with the median function is depicted in Figure 3a. Again, partial refinement was the optimal approach and computed up to 55 % less distances than conventional refinement. In case of 10-NN-queries with the minimum function (Figure 3b), partial refinement slightly improved the already very good results of conventional refinement. The increase in the number of required distance computations per added feature was surprisingly low for partial refinement (≈ 200). Note that the median ($m > 2$) and the minimum function do not fulfill the triangle inequality. Therefore, the Onion-tree frequently excluded objects that belonged to the correct query result.



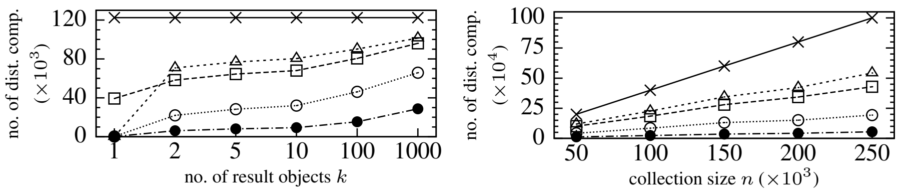
(a) Arithmetic mean (dist. comp.) **(b)** Arithmetic mean (search time)

Fig. 2. Search performance for 10-NN-queries with arithmetic mean (Caltech256).



(a) Median function **(b)** Minimum function

Fig. 3. Number of distance computations for 10-NN-queries (Caltech256).



(a) Arithmetic mean (Caltech256) **(b)** Arithmetic mean ($k = 10$, WEBUPV)

Fig. 4. No. of distance computations for kNN-queries with arithmetic mean ($m = 4$).

We conducted further experiments for other aggregation functions, like the maximum function, the geometric or the harmonic mean. However, these results are not shown as their behavior was mostly similar to the previous experiments.

6.3 Number of Result Objects and Collection Size

Figure 4a depicts the number of required distance computations of kNN-queries with the arithmetic mean for different numbers of result objects k. The Onion-tree and both filter refinement approaches were especially efficient for $k = 1$ because the query objects were elements of the collection. This allowed a very early termination of the search, as soon as the respective query object was seen the first time. However, partial refinement was the optimal approach for greater numbers of result objects k and constantly required approximately 65 % less distance computations than conventional refinement.

The impact of the collection size n on the number of needed distance computations is presented in Figure 4b. Subsets of the WEBUPV image collection were obtained by dividing it into chunks of 50,000 images each. While the number of needed distance computations increased linearly with the collection size for all approaches, partial refinement exhibited the overall lowest increase.

7 Conclusions and Outlook

This paper introduced partial refinement, a simple, yet efficient improvement of the filter refinement approach to similarity search with multiple features. Partial refinement progressively replaces partial distance bounds with exact partial distances, updates the aggregated bounds accordingly and checks if objects can be excluded.

Our experimental evaluation has shown that partial refinement is able to significantly reduce the number of required distance computations and search time in comparison to conventional refinement and other state-of-the-art techniques.

Future research will focus on the introduction of new strategies to determine the optimal order of partial distance computations. Adapting the computation order to the used distance and aggregation functions can further improve the search performance.

References

[1] Samet, H. *Foundations of Multidimensional and Metric Data Structures*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. San Francisco: Morgan Kaufmann Publishers Inc., 2005.

[2] Zezula, P., Amato, G., Dohnal, V., and Batko, M. *Similarity Search: The Metric Space Approach*. Vol. 32. Advances in Database Systems. Secaucus, NJ, USA: Springer-Verlag New York Inc., 2006, pp. 1–191.

[3] Böhm, K., Mlivoncic, M., Schek, H.-J., and Weber, R. “Fast Evaluation Techniques for Complex Similarity Queries”. In: *Proc. of the 27th International Conference on Very Large Data Bases*. VLDB 2001. San Francisco: Morgan Kaufmann Publishers Inc., 2001, pp. 211–220.

[4] Bustos, B., Keim, D., and Schreck, T. “A Pivot-Based Index Structure for Combination of Feature Vectors”. In: *Proc. of the 2005 ACM Symposium on Applied Computing*. SAC 2005. New York: ACM, 2005, pp. 1180–1184.

[5] Jagadish, H. V., Ooi, B. C., Shen, H. T., and Tan, K.-L. “Toward Efficient Multifeature Query Processing”. In: *IEEE Trans. on Knowl. and Data Eng.* 18 (2006), pp. 350–362.

[6] Zierenberg, M. and Bertram, M. “FlexiDex: Flexible Indexing for Similarity Search with Logic-Based Query Models”. In: *ADBIS 2013*. Ed. by Catania, B., Guerrini, G., and Pokorný, J. Vol. 8133. LNCS. Springer, Heidelberg, 2013, pp. 274–287.

[7] Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. “Searching in Metric Spaces”. In: *ACM Comput. Surv.* 33 (2001), pp. 273–321.

[8] Carélo, C. C. M., Pola, I. R. V., Ciferri, R. R., Traina, A. J. M., Jr., C. T., and Aguiar Ciferri, C. D. de. “Slicing the Metric Space to Provide Quick Indexing of Complex Data in the Main Memory”. In: *Inf. Syst.* 36.1 (2011), pp. 79–98.

[9] Fagin, R., Lotem, A., and Naor, M. “Optimal Aggregation Algorithms for Middleware”. In: *Proc. of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS 2001. New York: ACM, 2001, pp. 102–113.

[10] Zellhöfer, D. and Schmitt, I. “A Preference-Based Approach for Interactive Weight Learning: Learning Weights Within a Logic-Based Query Language”. In: *Distributed and Parallel Databases* 27 (2010), pp. 31–51.

[11] Bustos, B., Kreft, S., and Skopal, T. “Adapting Metric Indexes for Searching in Multi-Metric Spaces”. In: *Multimedia Tools Appl.* 58.3 (2012), pp. 467–496.

[12] Ciaccia, P. and Patella, M. “The M^2-tree: Processing Complex Multi-Feature Queries with Just One Index”. In: *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*. 2000.

[13] Hjaltason, G. R. and Samet, H. “Ranking in Spatial Databases”. In: *SSD 1995*. Ed. by Egenhofer, M. J. and Herring, J. R. Vol. 951. LNCS. Springer, Heidelberg, 1995, pp. 83–95.

[14] Griffin, G., Holub, A., and Perona, P. *Caltech-256 Object Category Dataset*. Tech. rep. 7694. California Institute of Technology, 2007.

[15] Villegas, M., Paredes, R., and Thomee, B. “Overview of the ImageCLEF 2013 Scalable Concept Image Annotation Subtask”. In: *CLEF 2013 Evaluation Labs and Workshop, Online Working Notes*. Valencia, Spain, 2013.