Scalable Similarity Queries over Complex Data

Dissertation Summary

Daniel Kocher
dkocher@cs.sbg.ac.at
University of Salzburg

Supervisor: Univ.-Prof. Dipl.-Ing. Nikolaus Augsten, Ph.D.
Co-Supervisor: Assoc.-Prof. Dr. Ana Sokolova

May 18, 2021

Abstract

In this dissertation, we study a particular class of queries in the context of database systems, namely similarity queries. Queries in database systems often use equality predicates to compare data items. Instead, a *similarity query* uses some notion of similarity to compare data items. The usage of similarity functions often increases the algorithmic complexity of such queries and prevents the application of standard query evaluation techniques (like hashing or sorting). As a result, similarity queries are often impracticable for realistic data. Therefore, novel efficient solutions for similarity queries that scale to large datasets must be developed. Specifically, we study two types of similarity queries. *Top-k subtree similarity queries* for trees aim to find and rank the k subtrees in a large document tree that are most similar to a given reference tree (the query tree). *Density-based clustering for sets* aims to partition a given collection of sets into dense regions that are separated by low-density regions. We pinpoint the problems of existing solutions and carefully design novel solutions to solve these problems efficiently. Furthermore, we empirically evaluate our solutions against the state of the art and provide in-depth discussions of the results.

This cumulative dissertation is organized as a collection of conference papers. It consists of an introduction and three chapters that are at the core of this dissertation. Each chapter tackles a specific type of similarity query. Chapter 2 [1] and Chapter 3 [2] are self-contained scientific publications that have been published at peer-reviewed venues. Chapter 4 extends the solution presented in Chapter 3 to multi-core environments and is being submitted to a peer-reviewed journal. The last chapter concludes this dissertation.

Alongside the development of new efficient solutions for similarity queries on an algorithmic level, we also contribute towards the reproducibility of data and experiments in the course of this thesis. First, we provide a reproducibility package for our ACM SIGMOD 2019 publication on top-k subtree similarity queries (Chapter 2). Our package has been awarded the ACM "Results Replicated" label by the ACM SIGMOD 2020 Reproducibility program committee. Second, the experiences in the course of this thesis led to a journal article [3] that highlights the importance of data reproducibility. The article points out that often not even the data that is used to run the experiments can be reproduced. Furthermore, we define a data reproducibility model called RPI that is based on three levels of data reproducibility (Raw data, Preparation instructions, and Input data), summarize our own experience, and exemplify our best practices. This was a collaborative work of the Database Research Group at the University of Salzburg and thus, is not part of this cumulative dissertation.

Introduction

Nowadays, database systems must manage continually and fast growing data volumes. Furthermore, database systems must be able to store and retrieve complex data items to meet the requirements of modern applications. Complex data items can stem from a wide range of areas, including hierarchically encoded data in bioinformatics or natural language processing, as well as set-encoded data to represent tweets, tags, or event logs. *Similarity queries* constitute one substantial class of queries in database systems. Traditional queries often evaluate equality predicates, i.e., data items are checked for their equality. Instead, similarity queries use predicates that compare data items based on some notion of similarity. Various similarity functions have been developed to assess the similarity for different data types like strings, trees, sets, or binary vectors. However, it is not straightforward to integrate similarity predicates into database operators efficiently. The similarity predicates introduce additional complexity into the query evaluation and often render standard techniques (like sorting or hashing) inapplicable. Similarity functions are often computationally expensive, and standard techniques typically fail to leverage specific optimizations for the underlying data type. Therefore, novel specialized solutions must be developed to support efficient similarity queries for large datasets.

In this cumulative dissertation, we focus on the development of efficient and scalable solutions for two popular types of similarity queries: (1) Top-k subtree similarity queries for trees, which aim to rank the k most similar subtrees in a large document tree with respect to a given reference tree, and (2) the density-based clustering of sets, which aims to identify clusters in a collection of sets based on the notion of density. Methodologically, we identify the problems of existing solutions, carefully design new data structures and algorithms to tackle these problems, implement our approaches and existing solutions in C++, empirically evaluate them against the state of the art, and discuss the outcomes thoroughly. Furthermore, we prove the correctness of our solutions and analyze them with respect to time and space complexity.

In the remainder of this dissertation summary, we summarize the core contributions of the respective topics in the cumulative dissertation. The last section concludes this summary.

Scalable Top-k Subtree Similarity Queries

Authors	Daniel Kocher and Nikolaus Augsten.
Title	A Scalable Index for Top-k Subtree Similarity Queries.
Venue	Int. Conf. on Management of Data (SIGMOD), Amsterdam. ACM, 2019.
Thesis Ref.	Chapter 2, Appendix A.

In this publication, we study so-called top-k subtree similarity queries. Given a large document tree, T, a *top-k subtree similarity query* retrieves the k subtrees in T that are most similar to a given query tree, Q. Specifically, we investigate this type of similarity query for ordered labeled trees and use the well-established tree edit distance (TED) to assess the similarity of two trees. In an ordered labeled tree, a strict, total order on the children of a node is enforced, and each node has a label that carries some information. The tree edit distance between two trees T_1 and T_2 is the minimum number of node edit operations that transform T_1 into T_2. Traditionally, the following three node edit operations are supported: (1) Rename the label of a node, (2) delete a node, and (3) insert a node. The tree edit distance requires cubic time and quadratic space in the number of tree nodes, hence a naive solution that computes the tree edit distance between the query tree Q and each single subtree of T are infeasible.

Previous solutions follow two different philosophies: (1) TASM-Postorder is the fastest solution without building an index structure. Instead, TASM-Postorder scans the entire document tree to answer a top-k subtree similarity query. Due to its index-free nature, TASM-Postorder has a very small memory footprint (independent of the size of the document tree) but runs slow compared to index-based solutions. (2) StructureSearch precomputes an index to answer top-k subtree similarity queries fast once the index is ready. However, StructureSearch suffers from a large memory footprint that is quadratic in the size of the document tree (for deep trees). Furthermore, StructureSearch must verify many subtrees to retrieve the final result, which results in high runtimes even for small values of k. Finally, StructureSearch is tailored towards XML documents and does not support index updates.

We propose a novel index-based solution called *SlimCone* that is based on the idea of a *candidate score*. Intuitively, subtrees with a high score are more likely to be close to the query tree in terms of tree edit distance. The candidate score is based on the tree labels, i.e., the score is high if the query tree and the subtree share many labels. Processing the subtrees in non-decreasing candidate score order provides two benefits: (1) It enables SlimCone to find good candidates fast. (2) Our algorithm can stop early because the candidate score implies a lower bound on the tree edit distance, i.e., none of the remaining subtrees will improve the ranking. Since the query is unknown when we build the index, we must establish the candidate score order on the fly at query time. We introduce a novel technique that builds inverted lists over the labels of the document tree. An inverted list of a particular label holds all subtrees that contain this label. Lists are then partitioned based on the sizes of the subtrees, and partitions are processed in the order of the best candidate score that can be found in this partition. Indexing all subtrees will however require space that is quadratic in the size of the document tree (similar to StructureSearch). To this end, SlimCone introduces an incrementally updatable, linear-space index structure that builds relevant partitions of the lists on the fly, and we show how to generate the partitions with minimum performance overhead. In our experimental evaluation, we observe that SlimCone clearly outperforms both TASM-Postorder and StructureSearch with respect to memory usage, index build time, number of verified subtrees, and query runtime.

For this publication, we created a reproducibility package that has been awarded with the "Results Replicated" label by the ACM SIGMOD 2020 Reproducibility program committee. The "Results Replicated" label indicates that the program committee has independently obtained experimental results that support the main results of the paper.

Density-Based Clustering for Large Collections of Sets

Authors	Daniel Kocher, Nikolaus Augsten, and Willi Mann.
Title	Scaling Density-Based Clustering to Large Collections of Sets.
Venue	Int. Conf. on Extending Database Technology (EDBT), Nicosia. OpenProceedings.org, 2021.
Thesis Ref.	Chapters 3 and 4.

Finding so-called *clusters* in a given collection of data items is an important operation in database systems. One popular clustering technique is the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm with numerous applications, for example, in biology, image processing, or process mining. The DBSCAN algorithm is based on the notion of density and identifies clusters as dense regions that are separated by low-density regions. Given a distance function between pairs of data points, the DBSCAN algorithm uses the concept of neighborhoods. For a particular data point r, the neighborhood contains all data points that are within a given distance (or radius). The clustering is then controlled using two input parameters: ϵ denotes the radius (i.e., the maximum distance) for other data points to be neighbors of r, and minPts specifies the minimum number of data points for a neighborhood to be dense. A data point with a dense neighborhood is called *core point*, a data point that is not core but in the neighborhood of a core point is a *border point*, and all other data points are *noise*.

The standard DBSCAN algorithm (randomly) picks a seed point r from the set of unvisited data points. If r is a core point, then a new cluster is formed by recursively expanding all neighbors of r (i.e., all neighbors of r are assigned to the new cluster and are expanded in the same manner in case they are core). A cluster is fully identified when all core points in a cluster have been expanded. The DBSCAN algorithm imposes a partial processing order on the neighborhood computations (neighbor by neighbor). To find the clusters fast, the neighborhoods must be computed efficiently. In our experiments, the neighborhood computation accounts for up to 99% of the overall runtime for some datasets.

In our publication, we consider density-based clustering for large collections of sets under Hamming distance constraints. Sets are often used as a representation of complex data items, and the Hamming distance can be used to assess the similarity of two sets based on the overlapping set elements (called tokens). The standard DBSCAN algorithm relies on a *symmetric index*, i.e., an index structure that reports the complete ϵ-neighborhood for a given set r. However, the most effective index structures for sets have been shown to be *asymmetric.* An asymmetric index assumes a processing order on the sets (typically based on the set sizes) and reports only the so-called *lookahead neighbors* of r, the neighbors that follow r in the processing order. Asymmetric indexes are clearly superior in terms of effectiveness compared to their symmetric counterparts, but are incompatible with the neighbor-by-neighbor order of the DBSCAN algorithm. We identify three issues that arise for the DBSCAN algorithm with asymmetric indexes: (1) The lookahead neighbors of data point r are insufficient to determine its core status. (2) We may wrongly classify a border point as noise because the lookahead neighbors only contain a specific part of the complete neighborhood. (3) Clusters may be disconnected because the DBSCAN algorithm expands all core points of the current cluster and relies on the complete neighborhoods (but we only see the lookahead neighbors).

Consequently, there are two options, none of which is satisfying: (1) *Sym-Clust* executes the DBSCAN algorithm with a symmetric index to retrieve the complete ϵ-neighborhoods, which has a small memory footprint but is slow because many sets must be evaluated. (2) *Join-Clust* leverages an asymmetric index in a self-join but must materialize the (quadratic-size) neighborhoods in main memory to execute DBSCAN on top, which is fast but infeasible for many datasets due to the large memory footprint.

We propose *Spread*, the first DBSCAN-compliant algorithm that uses asymmetric indexes in linear space. Spread is able to provide the efficiency of the join-based solution by imposing a specific processing order and leveraging the effective asymmetric index. At the same time, Spread avoids the materialization of the (quadratic-size) neighborhoods using two key data structures: *Backlinks* are dynamic collections of references that store enough information to build DBSCAN-compliant clusters (independently of the processing order) and require only $O(n)$ additional space (for a dataset with n data points). Furthermore, we use a disjoint-set data structure to maintain a graph of subclusters. The connected components in this graph then form DBSCAN-compliant clusters. Our experiments on 13 real-world datasets suggest that Spread is as fast as Join-Clust while being competitive with Sym-Clust in terms of memory.

The Spread algorithm has already shown to have industry impact: It has been implemented into the database back end at Celonis SE, a Munich-based company that develops the market-leading software in the process mining domain. For further practical impact, we have developed and evaluated an extension of Spread to multi-core environments, called *MC-Spread.* For a total number of $k+1$ concurrent threads, MC-Spread uses k threads to compute lookahead neighbors and one thread to build the clusters. The neighborhoods are stored in a shared array, and the neighborhood threads notify the clustering thread when a neighborhood is available. The clustering thread processes and frees the lookahead neighborhoods as they become available, respecting the user-defined processing order. We also introduce a memory-constrained version of MC-Spread that effectively bounds the memory the neighborhood threads are allowed to consume. This mitigates a (possibly) large memory footprint of MC-Spread, which may occur if the neighborhood threads allocate memory much faster than the clustering thread frees it. In our experiments, we evaluate MC-Spread against a multi-core extension of the materialization-based Join-Clust, and observe that MC-Spread scales better with the number of cores in terms of runtime. We also discuss dataset characteristics that affect the scalability of both solutions. We plan to submit the multi-core extension to a peer-reviewed journal in mid 2021.

Conclusions

Similarity queries are an important class of queries in database systems. A similarity query evaluates similarity predicates that typically compare data items based on some similarity function. Similarity functions often introduce additional complexity and render standard querying techniques (e.g., hashing) impracticable. In this dissertation, we studied two specific types of similarity queries, namely top-k subtree similarity queries for ordered labeled trees and density-based clustering for collections of sets. For each query type, we proposed specialized index structures and algorithms, which advance the state of the art.

A top-k subtree similarity query finds and ranks the k subtrees in a large document tree that are most similar to a given query tree. We focused on ordered labeled trees and used the tree edit distance as a black box to assess the similarity of two trees. Existing solutions are either slow with a small memory footprint or fast with a large memory footprint. We proposed *SlimCone*, the first updatable linear-space index structure for this query type. The index is based on inverted lists and allows us to retrieve promising subtrees first. We achieve linear space by building relevant parts of the lists on the fly (rather than materializing the full lists upfront). Our experiments confirm the superiority of our solution in terms of memory usage, index build time, query time, and number of verified subtrees.

Density-based clustering identifies high density regions as clusters that are separated by regions of lower density. The popular DBSCAN algorithm (randomly) picks unprocessed data points and recursively expands dense neighborhoods until a low-density neighborhood is found. We studied density-based clustering for collections of sets and the Hamming distance. For sets, the most effective indexes are asymmetric, i.e., only a specific part of the complete neighborhood is returned, the lookahead neighbors. Thus, asymmetric indexes cannot be readily combined with the DBSCAN algorithm (which imposes a partial neighbor-by-neighbor processing order). Instead, the DBSCAN algorithm must rely on symmetric indexes. Contrarily, a solution based on a self-join can use asymmetric indexes but must materialize the (quadratic-size) neighborhoods in main memory. We proposed *Spread*, the first DBSCAN-compliant linear-space solution that is able to leverage asymmetric indexes for sets. Spread imposes a specific processing order and only materializes one lookahead neighborhood at a time. We introduced backlinks and used a disjoint-set data structure to derive a DBSCAN-compliant clustering while leveraging asymmetric indexes. Our experiments suggest that Spread combines the best of the two worlds: It is competitive with the join-based solution in terms of runtime and retains the memory efficiency of the DBSCAN algorithm. We finally developed, implemented, and evaluated a multi-core extension of Spread.

References

[1] Daniel Kocher and Nikolaus Augsten. "A Scalable Index for Top-k Subtree Similarity Queries". In: *Proceedings of the 2019 International Conference on Management of Data.* SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1624–1641. ISBN: 9781450356435. DOI: 10.1145/3299869.3319892. URL: https://doi.org/10.1145/3299869.3319892.

[2] Daniel Kocher, Nikolaus Augsten, and Willi Mann. "Scaling Density-Based Clustering to Large Collections of Sets". In: *Proceedings of the 24rd International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021.* OpenProceedings.org, 2021. ISBN: 978-3-89318-084-4.

[3] Mateusz Pawlik et al. "A Link is not Enough - Reproducibility of Data". In: *Datenbank-Spektrum* 19.2 (2019), pp. 107–115. DOI: 10.1007/s13222-019-00317-8. URL: https://doi.org/10.1007/s13222-019-00317-8.