A Roadmap towards Declarative Similarity Queries

Nikolaus Augsten
University of Salzburg
Salzburg, Austria
nikolaus.augsten@sbg.ac.at

ABSTRACT

Despite many research efforts, similarity queries are still poorly supported by current systems. We analyze the main stream research in processing similarity queries and argue that a general-purpose query processor for similarity queries is required. We identify three goals for the evaluation of similarity queries (declarative, efficient, combinable) and identify the main research challenges that must be solved to achieve these goals.

1 INTRODUCTION

In a similarity query, two data objects "match" if they are similar. Similarity queries are required in scenarios where equality and exact matches are not effective, for example, when dealing with noisy data (e.g., in data analytics, ETL processes, data cleaning, and entity resolution), for detecting small differences between data objects (e.g., finding similar molecular structures in computational biology), for comparing complex objects (e.g., trees, graphs, or multimedia content), and for querying physical measurement data (e.g., time series, spatial objects, sensor data).

Similarity queries involve a *similarity function* defined on a specific data type and an *operator*. The similarity function assesses the similarity (or dissimilarity) between pairs of objects, e.g., the edit distance between strings or trees, Cosine similarity between sets, or Euclidean distance between vectors. The operator defines the input signature (e.g., lookup vs. join) and the pairs that qualify for the result set (e.g., top-k, range, or skyline query).

Database management systems (DBMS) for *exact queries* (i.e., predicates based on equality, less-than, or greater-than) have evolved into powerful and mature systems, which transparently and efficiently deal with data storage and querying. Unfortunately, this development has not happened for similarity queries. Applications that require advanced similarity features cannot rely on general-purpose systems that transparently handle data storage and querying. Instead, similarity queries must be dealt with in custom, ad hoc code. Writing custom code for similarity queries is expensive, requires advanced query processing skills, and often results in inflexible, hard-coded query plans. Changing the query beyond simple parameter settings requires additional programming efforts.

In this paper we argue that the integration of similarity queries into declarative DBMS (relational or non-relational) and the efficient processing in a systems context are the next challenges to be solved for similarity queries. In the past, the main research focus was on physical operators and access methods: new evaluation algorithms and index structures for specific combinations of operators, data types, and similarity functions were proposed. The evaluation of similarity queries in a larger systems context, however, has received little attention.

Developing a general-purpose query processor for similarity queries is challenging, both from a conceptual and a technical point of view. From a conceptual point of view, similarity queries pose a particular challenge. While the meaning of similarity is highly application dependent, the query interface should be general and serve a wide range of application needs. Note that application dependency is much less pronounced in equality queries: for most data types, the notion of equality is well defined and application independent.

From a technical point of view, many techniques that are effective for equality queries are not applicable to similarity queries. Techniques for equality queries often rely on exact matches or some ordering, which cannot be assumed between similar objects. An example is hashing, which leverages the fact that identical data values are hashed to the same bucket. This does not hold for similar values: they will typically be hashed to different buckets. Another example is sorting, which is not reliable for similar data items since even a small change may have a large impact on the position of the data item in the sort order. Further, little is known about the interaction of physical similarity operators with other, equality-based operators in the query context. This involves all aspects of query processing, including modeling similarity operators at the logical level (e.g., extending relational algebra), rewriting query plans, estimating the cost of similarity operators, and gracefully adapting to limited memory resources.

This paper suggests to depart from the main stream in similarity research with its narrow focus on individual physical operators and studies similarity queries from a broader systems perspective. The overall goal is to develop a deep understanding of all aspects of similarity queries that are required to build a general-purpose query processor for these queries.

2 DEC – DESIDERATA FOR SIMILARITY QUERIES

We identify three core requirements for a similarity query processing system: *declarative, efficient, combinable (DEC)*, i.e., declarative queries that combine equality and similarity predicates should be processed efficiently. We believe that all DEC requirements must be satisfied in a useful end-to-end system for similarity queries. We next discuss each of the three DEC requirements, which are orthogonal aspects of a query answering system.

Declarative. The queries should be declarative, i.e., the query describes *what* the answer to the query should look like rather than *how* the answer should be computed. A declarative approach allows flexible queries and a clearer separation between logical and physical layer. While the users express their queries at the logical level, the system must translate the query into a physical execution plan. Declarative data query languages are the predominant approach in the traditional relational model [9] with SQL (Structured Query Language) as a practical query language, but have also been applied to non-relational data models (e.g., XPath[1] for semi-structured data) and to cluster computing

[1]http://www.w3.org/TR/xpath-30/

systems with flexible data types (e.g., HiveQL [25] for analytic queries on large data clusters). Effective techniques for translating declarative queries to physical operators are required, and the resulting query plans must be optimized.

Efficient. The queries should be executed efficiently, i.e., efficient query plans that consider appropriate techniques for similarity operators should be constructed. Similarity predicates [13, 27] often involve complex and expensive functions, and a straightforward evaluation of the similarity predicate on each tuple is not feasible. A rich set of algorithms and access methods have been proposed to allow similarity queries to be processed efficiently. Such techniques are often based on an index or a filter/verify approach to reduce the number of expensive predicate evaluations [6, 11, 23]. Filters produce a set of candidates which contains false positives; in the verification step the false positives are removed. A widespread approach avoids the evaluation of the cross product in similarity joins by rewriting the join into an equality join on tokens, e.g., q-grams in a string similarity join [13]. When the similarity function is a metric, the triangle inequality and metric index structures can be leveraged [26]. A system for similarity queries should be able to effectively apply the efficient evaluation strategies that have been developed.

Combinable. Similarity and equality predicates should be arbitrarily combinable into complex queries. In useful queries, similarity predicates are embedded into a larger query context which includes a mix of equality and similarity predicates. In order to evaluate such a complex query efficiently, the query must be considered as a whole. It is not enough to process the similarity part independently from the equality part and then intersect (or union) the results. This may lead to very poor evaluation strategies since large intermediate results are produced. An example are conjunctive queries in which each individual predicate has weak selectivity (i.e., produces a large intermediate result set), but the conjunction of all predicates is strongly selective (i.e., the final result is small). A query processor for similarity queries should not be limited to the evaluation of the similarity predicate of the query, but should be able to evaluate both similarity and exact predicates alike.

3 SIMILARITY SUPPORT IN CURRENT SYSTEMS

Database Management Systems. DBMS typically offer basic support for similarity queries on strings, e.g., Soundex, a phonetic transcription of English surnames. More advanced similarity predicates are supported in the form of user defined functions (UDF). The UDF is used to evaluate the similarity predicate on a pair of attribute values. Unfortunately, the DBMS cannot produce efficient evaluation plans for queries with UDFs since they are a black box for the optimizer. UDFs are typically applied in a naive way (e.g., on each pair of tuples in a join [13]). Since the DBMS does not understand the properties of the similarity join, efficient filter/verify techniques (which have been proposed for similarity joins) or other optimizations cannot be leveraged. Overall, DBMS offer declarative, combinable similarity queries, but fail to process them efficiently.

Custom Software. Applications that require more sophisticated algorithms rely on custom software, e.g., as part of an entity resolution tool [10], at the back-end of a search form [18], or in some scientific application [7]. Due to the narrow focus and the predictable nature of the queries, the query plan is generated by hand and hard-coded, and appropriate algorithms and indexes are implemented. Hard-coded query plans are problematic since the quality of a query plan depends on the query parameters and the data distribution. Good query plans are particularly important for similarity queries, which often involve expensive predicates [14, 16, 27]. Extending custom software with new queries requires substantial programming efforts.

Integrating Similarity into DBMS. There have been several attempts to extend DBMS with similarity features. Barioni et al. [2] propose the SIREN system and an SQL extension to deal with similarity queries over multimedia and relational data. SIREN processes the similarity part outside the DBMS in a separate system and integrates the results in a second step. This separation substantially limits the options for efficient query plans since the similarity predicate cannot be freely moved in the query plan. Guliato et al. [15] propose an extension for PostgreSQL (an open source DBMS) for image retrieval. Similarly, the pg_similarity extension of PostgreSQL defines a set of similarity functions (e.g., edit and q-gram distance for strings). These approaches do not change the query processor, but are UDF-based and cannot leverage advanced algorithms, indexes, and optimization techniques. Silva et al. [21] integrate physical operators for similarity join and group-by into the core of PostgreSQL; the similarity operators are limited to numeric values. The Metric Similarity Search Implementation Framework (MESSIF) [3] is a library for object retrieval in metric space; it supports metric indexes and algorithms for range queries and top-k selection, but no declarative query interface or a query optimizer. An attempt to define an SQL-based query language for MESSIF has been reported, but query processing is not discussed.

Silva et al. [22] study the conceptual evaluation and query transformation rules for various types of similarity queries based on metric distances. In addition to the well-known ϵ-join (range distance join), the kNN-join (k nearest neighbor join), and kD-join (k-distance join) they also discuss join-around, a combination of range and nearest neighbor join. In terms of select queries, ϵ- and kNN-selection are discussed. Carey and Kossmann [5] and Bruno et al. [4] discuss the optimization of top-k queries.

The system closest to our vision is DIMA [24], which extends SQL with range queries over strings and sets. DIMA builds on Spark, supports distributed query evaluation, and uses a signature-based approach to distribute the query load and filter candidate matches. Compared to our vision discussed in the next section, the high-level similarity operators are not split into algebraic primitives, there is no metadata to select filters and transform the queries accordingly, and a high-level similarity operator is mapped one-to-one to the respective physical operator.

Other Systems. Entity resolution systems like NADEEF [10, 12] use similarity functions between individual attribute values to deal with noise in the data. Digital libraries deal with mixed objects (multimedia, text, 3D structures) with the goal of preserving digital objects, allowing users to enter new items, and accessing content. In both cases, the query patterns are hard-coded in the application, i.e., declarative queries are not supported.

Information retrieval systems like INDRI[2] or Lucene[3] store collections of documents (e.g, plain text, HTML, PDF) in files, build indexes over these files, and deal with stemming and stop word removal. Queries are phrases, possibly with wildcards, that

[2]http://www.lemurproject.org/indri
[3]http://lucene.apache.org

can be combined with boolean operators; the search can be limited to individual fields of a document (e.g., the title field). The query result is a list of documents, which is ranked by relevance. Similarity queries are supported at a very basic level. Lucene, for instance, supports the phonetic encodings Soundex and Metaphone, and edit distance selection. The queries in these systems are limited to selection and ranking; more complex query patterns (e.g., joins) are not supported.

4 ROADMAP

Challenges. We identify three challenges that must be addressed to build similarity queries into declarative database management systems: (1) A new, *minimal set of algebraic operators* for similarity queries must be defined. (2) *Dynamic rewriting:* metadata about eligible filters, indexes, and data transformations must be made available to the query optimizer. (3) A *uniform cost model* for physical similarity operators must be developed.

Minimal algebra. The goal is to develop a new algebra for similarity queries which extends relational algebra with a *minimal set of operator primitives* that is able to express a wide range of similarity operators.

Similarity (unlike equality) is not a binary predicate, and data items can be ranked by their "degree" of similarity w.r.t. some query. This gives rise to a large variety of new matching principles, for example, k-closest neighbor selection, similarity group-by, or join-around (k closest neighbors within a maximum distance range). Previous work, e.g., Silva et al. [22], defines an algebraic operator for each of these matching principles. We believe that this approach will not scale: (a) Each new operator requires deep changes in the system. (b) Query rewriting rules for each pair of operators must be defined, leading to a quadratic number of such rules. The key to success will be to establish a minimal, non-redundant set of primitives that are composed to express high-level operators. This will provide great flexibility in reordering queries, and introducing new high-level operators will not require changes in the algebra.

The new algebra should satisfy the following requirements. (a) *Minimal:* the new operators should be small, non-redundant primitives; the new algebra should cleanly separate two orthogonal concepts, which have often been mixed in previous work: the similarity function between two objects (e.g., edit distance) and the matching principle (e.g., top-k join). (b) *Expressive:* a wide range of similarity queries should be expressible in the new algebra; complex similarity operators (e.g., top-k join with edit distance), for which efficient implementations exist at the physical level, may be expressed by composing several of the new primitives at the logical level. (c) *Extendable:* Introducing new physical operators or query flavors at the user level should not require changes in the algebra. (d) *Transformable:* The new algebra should provide equivalence rules which allow the optimizer to reorder the logical operators in a flexible way. This is particularly important for small primitives since multiple primitives may compose a single physical operator.

Dynamic Rewriting. In order to expand queries with filters, metadata about similarity functions (e.g., edit distance), their relationship and properties (e.g., upper and lower bounds, metric properties), and applicable filters (e.g., q-grams for range joins) must be available. This metadata will be leveraged to dynamically produce query transformation rules. New filters can easily be introduced by updating the filter ontology: no changes in the optimizer are required.

The metadata stores properties of (a) similarity functions, (b) matching principles, and (c) filters, and their relationships. Similarity functions may satisfy metric properties (e.g., string edit distance) or even some L_i-norm (e.g., cardinality of set intersection, geographic distances). The relationships between similarity functions are guarantees like lower and upper bounds (e.g., the q-gram distance provides a lower bound for the more expensive edit distance). The filter ontology must further relate similarity function / matching principle pairs to eligible filter techniques. Some filters require data transformations, e.g., the computation of tokens [1], which should be specified in the metadata. The filter ontology must also provide information about the selectivity of filters, which will be used to model the query cost. Given a similarity predicate, the ontology should be able to derive all eligible filters (including necessary transformations and selectivity).

Expanding queries with filters leads to very different physical plans for a single logical query. The physical plans may involve techniques that have been developed by different communities, e.g., metric and token-based techniques. A uniform cost model is required to evaluate the plans. The cost model must consider the cost of similarity functions, which may be very expensive, filter selectivity, and the effect of filters on the data distribution.

Uniform cost model. A new cost model for physical similarity operators must be developed. The cost model should quantify the cost of different physical query plans, which result among others from introducing filters into the query plan or transforming data to the appropriate representation (e.g., string data may be transformed into tokens or signatures for filtering purposes). In the past, cost models for some individual operations (e.g., M-tree lookups [8]) have been developed. For other operations (e.g., set similarity joins [19]) experimental studies provide qualitative insights, but lack a model to predict the cost. Selectivity estimates [17, 20] are an important input for cost models, but selectivities are independent of physical operators. A quantitative assessment of the costs of all physical operators in the query is required. Computing comparable cost estimations is particularly challenging for approaches that have evolved in different communities, for example, edit similarity, token-based approaches, and metric techniques. The cost estimation will need to take into account the data distribution, any query parameters, the available resources, and the filter selectivities. The cost model should further integrate well with existing cost models for non-similarity operators since the overall query cost must be assessed.

Query processing. We envision the evaluation of a query that includes similarity predicates as illustrated in Figure 1: The parser generates a query tree that involves both standard relational operators and the new algebraic similarity primitives. Thereby, a high-level similarity operator like a top-k join or a skyline query will be represented by a number of low-level algebra operators. The query planner consults the similarity metadata to learn about eligible filter techniques like lower and upper bounds for the given similarity function and operator. Thereby, the query planner is not limited to the high-level operators in the original query. For example, there may only be filter information for nearest neighbor queries in the metadata, but the query is join-around [22] (which combines nearest neighbor and range join). The planner decomposes join around into algebraic primitives and tries to rearrange and match the primitives to known combinations in the metadata.

The query plans with filters will typically include additional algebra operators (representing the filter). Some filters will require

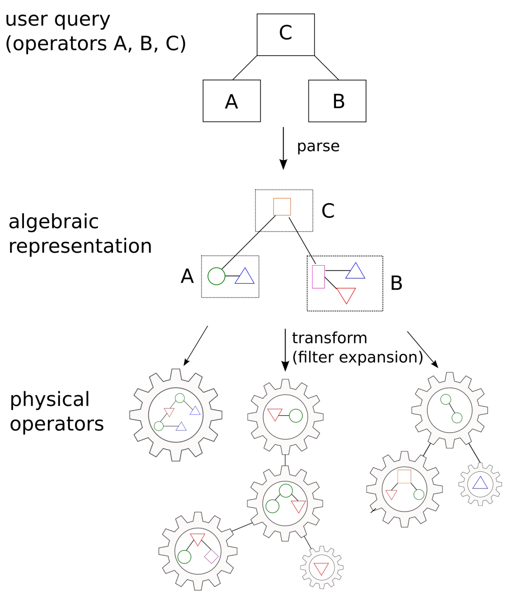

Figure 1: Interaction of system components.

on-the-fly index construction (e.g., prefix index for set similarity joins) or data transformations (e.g., tokens or signatures). The transformation rules for expanding the query plan with filters will be dynamically derived from the metadata. When new filters are registered in the metadata, they will trigger new query plans that apply these filters.

Finally, the cost of the query plans must be assessed. To this end, the algebraic operators must be arranged such that they match existing physical operators. Note that the logical query level (e.g., a query expressed in an SQL-like language) and the physical level are independent. There is no one-to-one match between the operators visible to the users and the physical operators actually implemented in the system. Rather, the operators in the user query are disassembled into algebra, and subtrees of the query plans are matched against existing physical operators.

5 CONCLUSION

This paper suggests to depart from the main stream in similarity research with its narrow focus on individual physical operators and studies similarity queries from a broader systems perspective. The overall goal is to develop a deep understanding of all aspects of similarity queries that are required to build a general-purpose query processor for these queries. The systems aspects discussed in this paper must be solved to enable wide applicability and impact of similarity queries in real systems. The main research challenges are the development of a minimal algebra for similarity queries, the design and querying of metadata regarding filter techniques for similarity queries, and the cost estimation for physical similarity operators.

A declarative interface will have a fundamental impact on the user interaction with similarity queries. Database users will no longer need to write ad-hoc code for evaluating similarity predicates. Instead, similarity predicates are expressed in a declarative way and are processed efficiently. Application developers will not need to bother about the details of similarity query processing. We expect a general-purpose query processor to trigger a wide adoption of similarity queries also in applications that so far could not afford the overhead of writing custom code.

Finally, a declarative similarity query processor will set new standards in the research community. New algorithms for physical operators must be evaluated against the query plans produced by the similarity-enabled optimizer, which can leverage a wide range of techniques, and dynamically adapts to query parameters and data distribution. Further, new algorithm proposals will not only be measured by their performance in an isolated setting, but also by their usefulness in a systems context.

Acknowledgements. This work was partially supported by the Austrian Science Fund (FWF): P 29859-N31.

REFERENCES

[1] Nikolaus Augsten, Armando Miraglia, Thomas Neumann, and Alfons Kemper. 2014. On-the-Fly Token Similarity Joins in Relational Databases. In *ACM SIGMOD*.
[2] Maria C. N. Barioni, Humberto Razente, Agma Traina, and Caetano Traina Jr. 2006. SIREN: A Similarity Retrieval Engine for Complex Data. In *Proc. Int. Conf. VLDB*.
[3] Michal Batko, David Novak, and Pavel Zezula. 2007. MESSIF: Metric Similarity Search Implementation Framework. In *Digital Libraries: Research and Development*. Vol. 4877. Springer Berlin Heidelberg.
[4] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2002. Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. *ACM Trans. Database Syst.* 27, 2 (2002).
[5] Michael J. Carey and Donald Kossmann. 1998. Reducing the Braking Distance of an SQL Query Engine. In *Proc. Int. Conf. VLDB*.
[6] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proc. Int. Conf. IEEE ICDE*.
[7] Davide Chicco and Marco Masseroli. 2015. Software Suite for Gene and Protein Annotation Prediction and Similarity Search. *IEEE/ACM Trans. on Computational Biology and Bioinformatics* 12, 4 (2015).
[8] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1998. A Cost Model for Similarity Queries in Metric Spaces. In *Int. Proc. ACM PODS*.
[9] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970).
[10] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: A Commodity Data Cleaning System. In *ACM SIGMOD*.
[11] Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2013. Top-k String Similarity Search with Edit-Distance Constraints. In *Proc. Int. Conf. IEEE ICDE*.
[12] Ahmed Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2014. NADEEF/ER: Generic and Interactive Entity Resolution. In *ACM SIGMOD*.
[13] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *Proc. Int. Conf. VLDB*.
[14] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. 2002. Approximate XML Joins. In *ACM SIGMOD*.
[15] Denise Guliato, Ernani V. de Melo, Rangaraj M. Rangayyan, and Robson C. Soares. 2009. POSTGRESQL-IE: An Image-Handling Extension for PostgreSQL. *J. Digital Imaging* 22, 2 (2009).
[16] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *PVLDB* 7, 8 (2014).
[17] Hongrae Lee, Raymond T. Ng, and Kyuseok Shim. 2011. Similarity Join Size Estimation Using Locality Sensitive Hashing. *PVLDB* 4, 6 (2011).
[18] Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. 2009. Efficient Type-Ahead Search on Relational Data: A TASTIER Approach. In *ACM SIGMOD*.
[19] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *PVLDB* 9, 9 (2016).
[20] Arturas Mazeika, Michael H. Böhlen, Nick Koudas, and Divesh Srivastava. 2007. Estimating the Selectivity of Approximate String Queries. *ACM Trans. Database Syst.* 32 (2007).
[21] Yasin N. Silva, Ahmed M. Aly, Walid G. Aref, and Per-Ake Larson. 2010. SimDB: A Similarity-Aware Database System. In *ACM SIGMOD*.
[22] Yasin N. Silva, Walid G. Aref, Per-Åke Larson, Spencer Pearson, and Mohamed H. Ali. 2013. Similarity Queries: Their Conceptual Evaluation, Transformations, and Processing. *PVLDB* 22, 3 (2013).
[23] Yasin N. Silva, Spencer Pearson, and Jason A. Cheney. 2013. Database Similarity Join for Metric Spaces. In *Proc. Int. Conf. Similarity Search and Applications*.
[24] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. 2017. Dima: A Distributed In-Memory Similarity-Based Query Processing System. *PVLDB* 10, 12 (2017).
[25] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-Reduce Framework. *PVLDB* 2, 2 (2009).
[26] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. 2006. *Similarity Search—The Metric Space Approach.* Advances in Database Systems, Vol. 32. Springer.
[27] Xiang Zhao, Chuan Xiao, Xuemin Lin, and Wei Wang. 2012. Efficient Graph Similarity Joins with Edit Distance Constraints. In *Proc. Int. Conf. IEEE ICDE*.