MSQL+: A Plugin Toolkit for Similarity Search under Metric Spaces in Distributed Relational Database Systems

Wei Lu†, Xinyi Zhang†, Zhiyu Shui†, Zhe Peng†, Xiao Zhang†, Xiaoyong Du†, Hao Huang‡, Xiaoyu Wang§, Anqun Pan§, Haixiang Li§

†*School of Information and DEKE, MOE, Renmin University of China, Beijing, China*
‡*School of Computer Science, Wuhan University, Wuhan, China*
§*Tencent Inc., China*
Contact:{luwei, duyong}@ruc.edu.cn, haohuang@whu.edu.cn

ABSTRACT

Similarity search is a primitive operation in various database applications. Thus far, a large number of access methods have been proposed to accelerate the similarity query processing. Nonetheless, these methods mostly focus on developing standalone systems by proposing new indices. Given the fact that existing RDBMS merely support traditional indices, it is of great necessity and practical importance to develop a standard RDBMS built-in index based approach to speeding up the query processing. In this demonstration, we introduce MSQL+, a plugin toolkit that enable users to answer similarity queries in metric spaces simply using standard SQL statements. This toolkit can help existing RDBMS to effectively and efficiently handle with big data due to the following three advantages. First, MSQL+ enables users to find similar objects by submitting SELECT-FROM-WHERE statements so that it can be easily integrated into existing RDBMS. Second, MSQL+ works in a more general data space. Objects of any type can be indexed by B$^+$-trees and the query processing can be boosted by using index seeks, as long as the similarity function is metric. Third, MSQL+ supports the parallelization of both pre-processing and query processing in distributed RDBMS.

PVLDB Reference Format:
Wei Lu, Xinyi Zhang, Zhiyu Shui, Zhe Peng, Xiao Zhang, Xiaoyong Du, Hao Huang, Xiaoyu Wang, Anqun Pan, Haixiang Li. MSQL+: A Plugin Toolkit for Similarity Search under Metric Spaces in Distributed Relational Database Systems. *PVLDB*, 11 (12): 1970-1973, 2018.
DOI: https://doi.org/10.14778/3229863.3236237

1. INTRODUCTION

Similarity search works as a primitive operation in many database applications, such as approximate string search in text databases [1], location based services in spatial databases [11], face recognition in multimedia databases [10], and structural motif discovery in protein databases [9]. Given a query object q and a collection of objects R, similarity search returns the set of objects from R whose distances to q are no greater than a user-defined threshold θ. A naive approach to answering similarity queries is to sequentially scan each object $r \in R$, and compute the similarity between r and q. As this naive approach is inefficient, a large number of access methods have been proposed to speed up the query performance. Nevertheless, these methods still suffer from either of the following three drawbacks.

• *Standalone.* Most of existing access methods focus on developing standalone systems by proposing new indices, such as M-Tree [4], D-Index [5], kd-tree, Quadtree, and Tries [2], to improve the efficiency. However, integrating these new indices into RDBMS is difficult, since existing RDBMS merely support built-in indices, typically including B$^+$-tree, R-tree, and hash index. Some other solutions [6, 12, 7, 3] index the data with B$^+$-trees and answer similarity queries by probing B$^+$-trees. Nonetheless, these solutions require new index probing mechanisms which are not supported by existing RDBMS unless new APIs are implemented. Furthermore, as discussed in [8], even if these solutions can be integrated into RDBMS with newly introduced APIs, their performance may be degraded to table scans.

• *Working in restricted data spaces.* Many existing access methods try to improve their efficiency with new pruning rules. Nevertheless, each of these methods works in a specific data space, and extending them to other data spaces is often infeasible. For example, methods that are proposed to answer string similarity queries [12, 7] work in text spaces only, and cannot be extended to Euclidean spaces or protein spaces. As a primitive operation, a similarity search approach should be general enough to deal with various database applications in RDBMS.

• *Running on centralized systems only.* In the era of big data, it is imperative to utilize distributed systems to manage the ever-increasing data. In many big internet enterprises nowadays like Tencent, data are split across multiple compute nodes, and both OLTP/OLAP queries are executed over the distributed data directly. Hence, using distributed similarity processing approaches is an inevitable trend, while existing methods work in centralized systems only.

To avoid the above three drawbacks, we propose MSQL+, a plugin toolkit based on our previous work [8] that is able to

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236237

answer similarity queries in distributed RDBMS fully using SQL statements. MSQL+ consists of two main phases.

- *Index building.* As long as the similarity function is metric, MSQL+ generates pair-wise comparable signatures for objects, and builds B^+-trees to index objects. Objects with signatures within a set of intervals are taken as candidates for similarity search.
- *Query processing.* MSQL+ enables users to find similar objects by merely submitting SELECT-FROM-WHERE statements with two predicates. One predicate involves in the similarity function which is implemented as an user-defined function. The other predicate specifies signatures in a certain set of ranges. The latter predicate triggers multiple index seeks to filter out false positives, while the former predicate verifies the candidates.

Compared with existing solutions, MSQL+ has the following three advantages. (1) *MSQL+ answers similarity queries simply using SQL statements.* (2) *MSQL+ works in a more general data space.* (3) *MSQL+ can run on both centralized and distributed RDBMS.*

2. TECHNICAL BACKGROUND

2.1 Similarity Search in Metric Spaces

MSQL+ adopts the divide-and-conquer paradigm to process similarity queries. The rationale of MSQL+ is to first select m objects as pivots and assign each object $r \in R$ to one and only one pivot according to a certain strategy (e.g., the pivot leading to a minimal distance). Then, the data space is split into m disjoint partitions. Let $\mathbb{P}$ be the set of selected pivots. $\forall p_i \in \mathbb{P}$, P_i^R denotes the partition whose objects take p_i as their pivot. The distance $|r, p_i|$ in each partition P_i^R is also maintained ($r \in P_i^R$). Then, similarity search is conducted by checking each partition P_i^R individually. Following the filter-and-verify paradigm, according to Theorem 1, objects in P_i^R with their distances to p_i within interval $[LB_i, UB_i]$ are taken as candidates. In this way, all the candidates are verified and similar objects are obtained.

THEOREM 1. *Given a partition P_i, $\forall r \in P_i$, the necessary condition for $|q, r| \leq \theta$ is as follows.*

$$LB_i = |p_i, q| - \theta \leq |p_i, r| \leq |p_i, q| + \theta = UB_i.$$

2.2 Pivot Selection

Theorem 1 shows that the search range $[LB_i, UB_i]$ for each P_i^R relies on pivot p_i. Therefore, it is necessary to select a set of good pivots which can enhance the pruning power of Theorem 1. So far, there are four types of commonly used methods for pivot selection.

- *Random* method randomly extracts a set of objects from the collection of objects R, and takes them as pivots.
- *MaxVariance* method selects pivots from R so that the variance of objects in R w.r.t. the pivots is maximized.
- *MaxProb* method selects pivots from R so that the expected number of objects taken as candidates is minimized.
- *Heuristic* method works like k-means and adopts a heuristic approach to selecting pivots so that the overall distances among queries to the pivots are approximately minimized.

2.3 Processing similarity queries in RDBMS

We assume that there exists an M-attribute schema for data set R, in which the similarity is measured on a subset of M attributes (denoted as $\mathcal{A}$:$\{A_1, A_2, ..., A_N\}$ $N \leq M$). Given $r \in R$, let $r[\mathcal{A}]$ be the set of attribute values of r over $\mathcal{A}$. To support B^+-tree boosted similarity queries, MSQL+ executes in two stages, namely (1) the offline index building and (2) the online query processing. The first stage generates a B^+-tree index over attributes $\mathcal{A}$, and the second stage runs the query processing using index seeks.

2.3.1 Index Building

We build the index over the attributes with two requirements. First, the attribute values must be comparable so that it is able to be indexed by B^+ trees. Second, it is able to figure out candidates for the similarity queries by simply comparing the attribute values. Apparently, by building the index with the above two requirments, it is able to answer similarity queries by probing the index. For this purpose, we propose a signature generation scheme with which we generate a signature $S(r[\mathcal{A}])$ for each record $r \in R$. Recall that in Section 2.1, R is split into $|\mathbb{P}|$ partitions, i.e., $R = \bigcup_{i+1}^{|\mathbb{P}|} P_i^R$, where P_i^R denotes the ith partition with p_i as the pivot. Given a partition P_i^R in R, $\forall r \in P_i^R$, $S(r[\mathcal{A}])$ is defined as a pair shown below:

$$S(r[\mathcal{A}]) = \langle i, |r, p_i| \rangle \quad (1)$$

where i is the partition ID and $|r, p_i|$ is the distance between r and p_i. Given two signatures $\langle i, d \rangle$ and $\langle j, d' \rangle$, the comparison rule is as follows.

$$\begin{cases} \langle i, d \rangle > \langle j, d' \rangle, & if\ i > j\ or\ (i = j\ and\ d > d'), \\ \langle i, d \rangle = \langle j, d' \rangle, & if\ i = j\ and\ d = d', \\ \langle i, d \rangle = \langle j, d' \rangle, & otherwise \end{cases}$$

Instead of directly building a B^+-tree over $\mathcal{A}$, we append a new attribute I (i.e., signatures) to R, build a B^+-tree over I. $\forall r \in P_i^R$, we update $r[I]$ to its correspondingly signature $\langle i, |r, p_i| \rangle$. Clearly, our index satisfies the above two requirements shown below. (1) $\forall r_1, r_2 \in R$, $S(r_1[\mathcal{A}])$ and $S(r_2[\mathcal{A}])$ are comparable. (2) Records with their signatures in an interval, i.e., $\forall r \in R, S(r[\mathcal{A}] \in [LB, UB])$, are taken as candidates. Additionally, because there is no intersection between ranges of signatures from different partitions, duplicated traversal of the index is avoided during the candidate identification.

2.3.2 Query Processing

We answer similarity queries in RDBMS using index seeks. First, we implement the commonly used similarity functions as user-defined functions. A distance function denoted as DIST($r[\mathcal{A}], q[\mathcal{A}], \theta$) returns true when the distance between $r[\mathcal{A}]$ and $q[\mathcal{A}]$ does not exceed θ. A naive approach to answering similarity queries using SQL is as follows.

```
SELECT   R.A_1,...,R.A_N
FROM     R
WHERE    DIST(r[A], q[A], θ)
```

Second, we apply index seeks to find candidates, which are then verified by computing the similarity. Based on Theorem 1, for each partition P_i^R, we can figure out an interval $[LB_i, UB_i]$ ($LB_i = \langle i, |r, p_i| - \theta \rangle$, $UB_i = \langle i, |r, p_i| + \theta \rangle$), and records with signatures in this interval are considered as candidates. Towards this, we maintain a list of

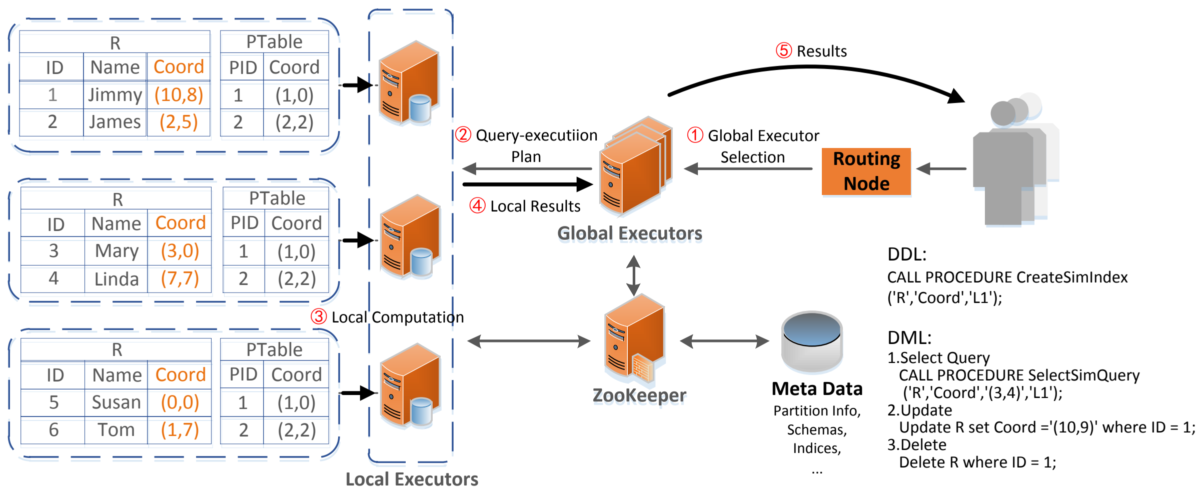

Figure 1: Overview of MSQL+ implemented on TDSQL

LB_i, UB_i for each pivot $p_i \in \mathbb{P}$ in a temporary relation, namely $PivotsRangeSet$, and process similarity queries using the following SQL statement.

```
SELECT   R.A_1,...,R.A_N
FROM     R, PivotsRangeSet PRS
WHERE    I BETWEEN PRS.LB and PRS.UB AND
         DIST(r[𝒜], q[𝒜], θ)
```

Since relation $PivotsRangeSet$ is small, the query optimizer always invokes index seek to identify candidates and then refines them by the filter DIST.

3. OVERVIEW OF MSQL+

We implement MSQL+ on top of Tencent TDSQL [1], a distributed RDBMS [2]. We suppose that relation R is split and stored across multiple data nodes, pivots are maintained in a table, namely PTable. Now our objective is to build a distributed B$^+$-tree index over $\mathcal{A}$ of R and use the index to speed up similarity queries.

3.1 System Architecture

As a typically distributed RDMBS, TDSQL mainly consists of four components that are shown in Figure 1. **L**ocal executors, which also work as data nodes, are responsible for storing/fetching tuples locally, shuffling/receiving tuples to/from other local executors, and executing local computation, like join, filter, etc. **Z**ookeeper maintains meta data, such as schemas, indices, relation partitioning information, etc. By taking the user-submitted SQL statements as the input, **G**lobal executors analyze the statements, generate and execute the query-execution plan based on the meta data. Following the execution plan, global executors coordinate local executors to accomplish local computation and shuffle data among them if necessary. **T**he routing node accepts users' requests and selects a global executor to answer the request by taking into consideration workload balance.

[1]http://tdsql.org
[2]Note that MSQL+ can be integrated into other distributed RDBMS in a similar way.

MSQL+ is implemented as a pluggable toolkit with multiple user-defined functions and stored procedures, and hence it can be seamlessly integrated into TDSQL. In MSQL+, the only meta data is **P**Table that is a collection of selected pivots. As presented in Section 2, PTable helps build the index and generate intervals for index seeks. Similar to other meta data, PTable is maintained in Zookeeper, and synchronized to all local executors. To help boost similarity queries, MSQL+ provides two stored procedures to build indices and answer queries, respectively.

3.2 Index Building

We provide a stored procedure *CreateSimIndex* encapsulated with DDL statements to build indices. *CreateSimIndex* takes relation name, attributes, similarity function name, pivot selection strategy, and pivot number as input parameters. We set defaulted values for the latter two parameters which can be omitted in the procedure. Either pivot selection, signature generation, or index building is implemented as user-defined functions. A global executor coordinates the local executors to build indices. First, each local executor is scheduled to select a certain number of pivots. The global executor aggregates all pivots, which are then stored in Zookeeper and synchronized to local executors. Second, based on these pivots, local executors are requested to generate signatures for the local records over the corresponding attributes, and build the local index once the generation stage completes.

3.3 Query Processing

We provide a stored procedure *SelectSimQuery* to process similarity queries. *SelectSimQuery* takes relation name, query, attributes, similarity function name, θ as input parameters, and query results as output parameters. When a user calls the procedure *SelectSimQuery*, the routing node selects one global executor, which translates the procedure into a set of SQL statements. The global executor executes the execution plan, coordinates local executors to first create a temporary relation $PivotsRangeSet$ maintaining the intervals LB_i, UB_i (shown in Section 2.3.2) for each partition P_i^R, and then do similarity query processing locally. In this way, each similarity query processing is fully parallelized.

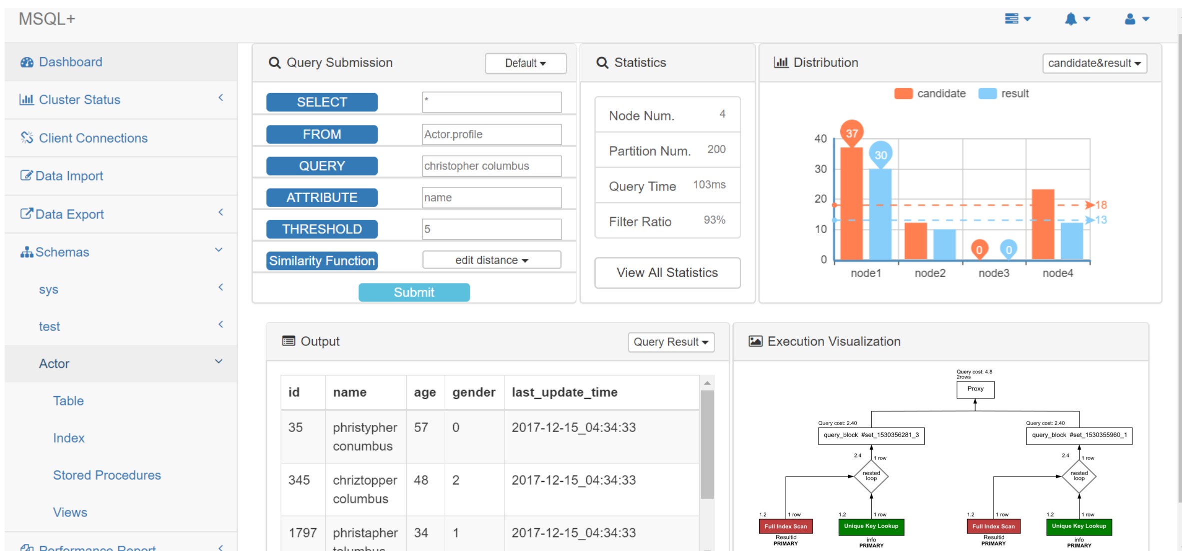

Figure 2: GUI of MSQL+

4. DEMONSTRATION

We demonstrate MSQL+ over text, Euclidean, and protein spaces, through the web interface shown in Figure 2. In this demonstration, the main modules are listed as follows.

• *Query processing.* Users can navigate relations under the schema directory. By clicking a relation, the query interface shows on the right of the window. After users type the input parameters and click the submit button in the query submission panel, the interface returns the statistics of the query execution, query results, distributed execution plan visualization, as well as the statistics of the query results among local executors. These functions are designed to help users to better understand how MSQL+ runs in distributed RDBMS, whether the execution over local executors is skewed, and the other execution information.

• *Index building.* Under the directory of a relation, users can click the index link to build or browse indices over the current relation. To build the indices, users need to input the attribute(s), similarity function, pivot selection strategy and pivot number (the latter two are optional), ending by clicking build button. Similar to operating result in query processing, the interface returns both the overall, and local statistics of the execution.

• *Others.* Other necessary functions include client connection, cluster monitoring, data import and export, user-defined function, stored procedure management, etc.

5. CONCLUSION

In this demonstration, we present MSQL+, a pluggable toolkit that enables RDBMS to process similarity queries using SELECT-FROM-WHERE statements. As a systematic solution, MSQL+ works in a more general data space. As long as the similarity function is metric, objects of any type can be indexed and the query process can be boosted using index seeks. MSQL+ supports queries in distributed systems, and is proposed as a complementary to existing RDBMS for the big data era.

Acknowledgements. This work was supported by the National Key Research and Development Program of China (No. 2018YFB1004401)Beijing Municipal Science and Technology Project (No.Z171100005117002), and Tecent Research Grant (RUC).The National Natural Science Foundation of China in part supports Wei Lu's work under Grant No. 61502504, U1711261 and 61702432, Xiaoyong Du's work under Grant No. 61732014 and U1711261, and Hao Huang's work under Grant No. 61502347. Hao Huang is the corresponding author.

6. REFERENCES

[1] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

[2] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, pages 604–615, 2009.

[3] L. Chen, Y. Gao, X. Li, C. S. Jensen, and G. Chen. Efficient metric indexing for similarity search. In *ICDE*, pages 591–602, 2015.

[4] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *PVLDB*, pages 426–435, 1997.

[5] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools Appl.*, 21(1):9–33, 2003.

[6] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[7] W. Lu, X. Du, M. Hadjieleftheriou, and B. C. Ooi. Efficiently supporting edit distance based string similarity search using B^+-Trees. *IEEE Trans. Knowl. Data Eng.*, 26(12):2983–2996, 2014.

[8] W. Lu, J. Hou, Y. Yan, M. Zhang, X. Du, and T. Moscibroda. MSQL: efficient similarity search in metric spaces using SQL. *VLDB J.*, 26(6):829–854, 2017.

[9] M. S. Waterman. *Introduction to computational biology - maps, sequences, and genomes: interdisciplinary statistics.* CRC Press, 1995.

[10] A. Yoshitaka and T. Ichikawa. A survey on content-based retrieval for multimedia databases. *IEEE Trans. Knowl. Data Eng.*, 11(1):81–93, 1999.

[11] R. Zhang, P. Kalnis, B. C. Ooi, and K.-L. Tan. Generalized multidimensional data mapping and query processing. *ACM Trans. Database Syst.*, 30(3):661–697, 2005.

[12] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, pages 915–926, 2010.