The VLDB Journal (2017) 26:829–854
DOI 10.1007/s00778-017-0481-6

REGULAR PAPER

MSQL: efficient similarity search in metric spaces using SQL

Wei Lu[1] · Jiajia Hou[1] · Ying Yan[3] · Meihui Zhang[2] · Xiaoyong Du[1] · Thomas Moscibroda[4]

Received: 27 November 2016 / Revised: 25 August 2017 / Accepted: 7 September 2017 / Published online: 6 October 2017

Abstract Similarity search is a primitive operation that arises in a large variety of database applications. Typical examples include identifying articles with similar titles, finding similar images and music in a large digital object repository, etc. While there exist a wide spectrum of access methods for similarity queries in metric spaces, a practical solution that can be fully supported by existing RDBMS with high efficiency still remains an open problem. In this paper, we present MSQL, a practical solution for answering similarity queries in metric spaces fully using SQL. To the best of our knowledge, MSQL enables users to find similar objects by submitting SELECT-FROM-WHERE statements only. MSQL provides a uniform indexing scheme based on a standard built-in B^+-tree index, with the ability to accelerate the query processing using index seek. Various query optimization techniques are incorporated in MSQL to significantly reduce CPU and I/O cost. We deploy MSQL on top of PostgreSQL. Extensive experiments on various real data sets demonstrate MSQL's benefits, performing up to two orders of magnitude faster than existing domain-specific SQL-based solutions and being comparable to native solutions.

Keywords Similarity search · Metric space · Query optimization · SQL-based · RDBMS

✉ Xiaoyong Du
duyong@ruc.edu.cn

Wei Lu
lu-wei@ruc.edu.cn

Jiajia Hou
houjiajia@ruc.edu.cn

Ying Yan
ying.yan@microsoft.com

Meihui Zhang
meihui_zhang@yeah.net

Thomas Moscibroda
moscitho@microsoft.com

1 DEKE, MOE and School of Information, Renmin University of China, Beijing, China

2 School of Computer Science & Technology, Beijing Institute of Technology, Beijing, China

3 Microsoft Research, Beijing, China

4 Microsoft Azure, Redmond, WA, USA

1 Introduction

Similarity search is a primitive operation that is at the heart of a wide spectrum of database applications, including face recognition, fingerprint matching in multimedia databases [1,33], location based services in spatial databases [35], error checking in text retrieval [3], pattern recognition (e.g., DNA or Protein sequences) in computational biology [29], etc. A thorough study on similarity search can be found in the survey [7,17]. Given a query object q and a collection of objects R, similarity search finds the set of objects from R whose distances to q are no larger than a user-defined threshold θ.

A naive approach to processing similarity queries is to perform a sequential scan over the entire data set R, which results in a query processing cost that increases linearly with the cardinality of R. Over decades, there has been significant interest in designing better access methods to support efficient similarity search. As shown in Fig. 1, these solutions can be categorized according to their access methods and system support.

A majority of solutions—**native Solutions**—focus on building new indexing techniques as stand-alone systems to improve efficiency. These solutions are optimized for their own specific applications and according to their own cost

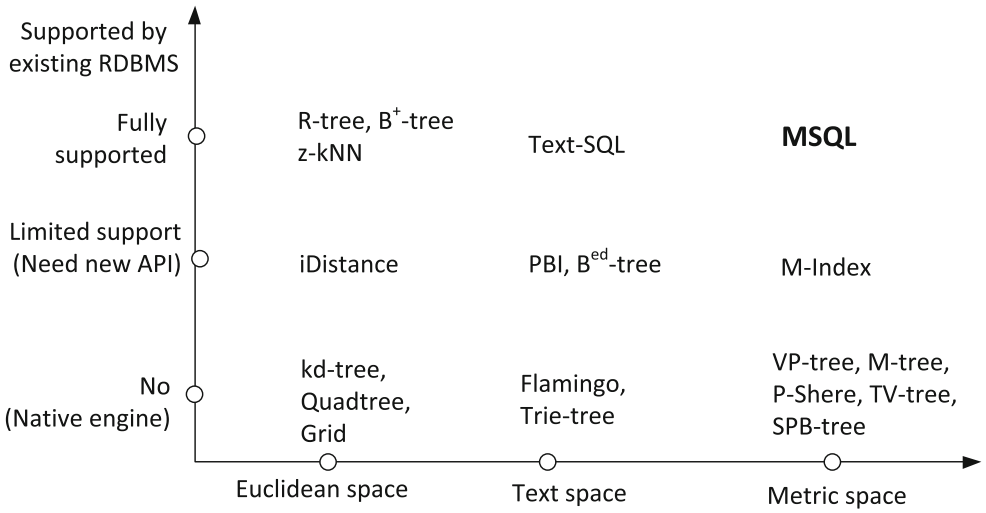

Fig. 1 Existing solutions in relation to **MSQL**

models which are typically hard to generalize. Examples in *metric spaces* include the VP-Tree [32] and its variant [5], M-Tree [9] and its variants [2,19], D-Index [11], P-Sphere Tree [14], TV-Tree [22], etc. In the more restricted *Euclidean spaces*, there are kd-tree, Quadtree, Grid, X-tree, and solutions based on Tries and inverted indexes [4,15,21,30] have been proposed under the edit distance metric in *text spaces*.

There also exist some solutions that index the data using the B^+-trees and answer similarity queries by probing B^+-trees. Examples include the UB-tree, iDistance [18,34] for *Euclidean spaces*, and B^{ed}-tree [36], PBI [23] for *text spaces* and M-Index [25], SPB-tree [8] for *metric spaces*. Nevertheless, these solutions cannot be effectively integrated into RDBMS. The reason is twofold. First, these solutions require new index probing mechanisms which are not supported by existing RDBMS unless new APIs are implemented (details are presented in Sect. 2.3). However, integrating new APIs into open source RDBMS is not trivial and it is infeasible to add new APIs to commercial RDBMS. Second and worse yet, as we shall discuss in Sect. 3, even when integrating these solutions into RDBMS with newly introduced APIs, their performance can be degraded to table scans, canceling the effect of index construction.

The alternative to the above solutions are **SQL-based solutions**. An SQL-based solution is attractive due to its high portability across various commercial and open source RDBMS. Besides, it automatically inherits features and performance enhancements that are available by default in the relational world, yet are often missing in native solutions (e.g., transactions, access control and replication, etc.). Decades of research and development in RDBMS have led to standardized querying interfaces and many sophisticated optimizations which could be leveraged in designing the solution to similarity search. The R-tree and B^+-tree are standard built-in indices used in existing RDBMS to serve similarity search in low-dimensional spaces. For instance, z-kNN [31] processes k nearest neighbor queries over *Euclidean spaces* using SQL by transforming every multi-dimensional point into a one-dimensional value (z-value), which is then indexed using a B^+-tree. Text-SQL [15] is the only known method to process *string* similarity search under *edit distance metric* using SQL. Yet, finding an efficient SQL-based solution for *general metric spaces* still does not exist and has remained an open problem for a long time. This is the problem we address in this paper.

Designing SQL-based solutions for efficient similarity search in metric spaces is challenging for several reasons. First, the proposed solution must be generic and portable across RDBMS, i.e., the solution must be based on standard built-in index structures (typically including B^+-tree, R-tree, and hash index) and index probing mechanism supported by existing RDBMS. To the best of our knowledge, there do not exist any precise solutions that answer similarity queries in metric spaces based on the R-tree and hash index, while B^+-tree-based solutions cannot be integrated into existing RDBMS unless new APIs are provided, which is obviously infeasible in commercial RDBMs. Second, it is hard to guarantee the efficiency and scalability of the SQL-based solution, especially when data volume is large and when data are frequently deleted/inserted/updated. How to design optimizations based on the features of optimized SQL operators toward a solution with high efficiency in a dynamic environment is a challenging issue.

In this paper, we propose **MSQL**, a new solution to similarity search in metric spaces using SQL, which can be fully supported by existing RDBMS. By implementing similarity functions as UDFs, similarity queries can be expressed using standard SELECT-FROM-WHERE statements. Suppose that each object $r \in R$ has M attributes $\mathcal{R} : \{A_1, A_2 \ldots, A_M\}$ and the first N attributes $\mathcal{A} : \{A_1, A_2 \ldots, A_N\}$ are used for

similarity measurement, i.e., r is considered a similar object to a query q if DIST$(r[\mathcal{A}], q[\mathcal{A}], \theta)$ returns true (if and only if the distance between r and q does not exceed θ), where DIST is the distance function, $r[\mathcal{A}]$ (or $q[\mathcal{A}]$) is the value list of object r (or q) over attribute list $\mathcal{A}$. We implement the distance function as a UDF and hence the similarity queries can be answered using SQL as follows:

```
SELECT   R.A1, ..., R.AN
FROM     R
WHERE    DIST(R[A], q[A], θ);
```

To evaluate this query, a vanilla query engine in RDBMS would essentially have to conduct a sequential scan (i.e., table scan) of the relation R and apply the UDF comparison as a post-processing filter. As frequently pointed out in the literature, this makes the cost of sequential scan prohibitive when the data set is large or the distance computation is expensive. For these reasons, we target a solution with index seek. In this paper, we propose a B^+-tree-based index structure to process similarity queries which is generally and efficiently supported in existing RDBMS.

The basic idea of our solution is to introduce an additional column I to the original table and build a B^+-tree index over column I. To support efficient similarity search, values of attribute I must satisfy at least two requirements: (1) any two rows of I values are comparative, and hence it is possible to build the B^+-tree index, and (2) I should encode the relative distance information of $\mathcal{A}$ values so that it is possible to prune the objects that are not result candidates based on the attribute values over I, i.e., we can employ the index seek instead of table scan to process similarity queries (see Sect. 3 for how to design such an index column I). With column I, we can use the following statement to conduct similarity search.

```
SELECT   R.A1, ..., R.AN
FROM     R
WHERE    LOCATE(R.I, @ranges) AND
         DIST(R[A], q[A], θ);
```

In the above statement, we introduce another UDF LOCATE $(R.I$, @ranges) as a filter, with which, objects whose values of attribute I do not fall into the range(s) @ranges are pruned. In other words, LOCATE works as an index probing function to filter out the unqualified objects. Objects which pass the LOCATE filter are further refined by function DIST. For example, filter $1 \leq R.I \leq 10$ will invoke an index seek to figure out candidates with $R.I$ in range [1, 10], which are then refined by function DIST. Intuitively, when the number of search ranges is small, the above SELECT-FROM-WHERE SQL statement will always invoke index seek to identify candidates. Nevertheless, as pointed out in existing work [23,24], to enable fast query processing against a data set with millions of objects, it is often necessary to have hundreds or thousands of search ranges in WHERE conditions, making the query optimizer resort to table scan to identify the candidates. To address this issue, we propose three optimization techniques. First, we propose a query rewriting technique that transforms the user-submitted SQL statement to a two-table join-based SQL statement. For the rewritten and join-based SQL statement, the query optimizer always invokes index seek to identify candidates and then refines each of them in turn by function DIST. Second, to improve the query performance, we propose a new pruning technique, under which MSQL is able to reduce both the number of search ranges @ranges and the upper bound of each unpruned search range while keeping the lower bound invariant. Third, we further investigate the problem of minimizing the number of objects, taken as candidates, residing in the unpruned search ranges under the new pruning rule.

MSQL eliminates the limitations of existing indexing methods and supports multiple types of similarity queries which fall into the following categories using a single SQL statement: (1) different similarity functions over the same column, (2) the same similarity function over different columns, (3) different similarity functions over different columns. In summary, our contributions are as follows:

- To the best of our knowledge, MSQL is the first comprehensive solution for similarity queries in *metric spaces* using SQL. MSQL is able to be deployed in any of existing RDBMS with the standard built-in B^+-tree index.
- MSQL provides a generic framework through which it enables objects of any data type in metric spaces to be indexed using a B^+-tree. We propose a query rewriting technique to make sure that the RDBMS query optimizer always employs index seek to process similarity queries.
- To improve query performance, we propose a new pruning rule that is able to reduce the number of predicates in WHERE conditions of the SQL statement. Further, we investigate the problem of minimizing the number of candidates in similarity search. We prove the problem is NP-hard and propose a heuristic approach.
- We present how to do insert, delete, and update operations in MSQL. Brought by the benefit of RDBMS, MSQL is able to perform similarity queries while other data manipulation operations including insert, delete, and update, are executed simultaneously. To the best of our knowledge, this is the first work to investigate the problem of processing similarity queries in a dynamic environment.
- We deploy MSQL on top of PostgreSQL and conduct extensive experiments on various real data sets under both static and dynamic environments. These experi-

Table 1 Symbols and their definitions

Symbol	Definition
$\mathcal{D}$	A metric space
R	The input dataset in $\mathcal{D}$
$dist(r, r')$	The distance between two objects r and r' in $\mathcal{D}$
$\|r, r'\|$	A simple representation of $dist(r, r')$
$\mathcal{R}$	The M-attribute schema storing R into a relation
$\mathcal{A}$	The first N attributes of $\mathcal{R}$ used for similarity search
q	A query object in $\mathcal{D}$
θ	A query threshold
$\mathbb{P}$	A set of pivots
p_i	A pivot in $\mathbb{P}$
P_i^R	The partition of R that corresponds to p_i
LB_i	The lower bound of search range for partition P_i^R
UB_i	The upper bound of search range for partition P_i^R

ments demonstrate the benefits of MSQL, outperforming existing techniques by up to two orders of magnitude.

The rest of the paper is organized as follows. Section 2 formalizes the problem, and discusses related work. Section 3 gives an overview of the MSQL framework. Section 4 presents the optimizations. Section 5 reports the experimental results and Sect. 6 concludes the paper.

2 Preliminaries

In this section, we formalize the problem, review the reference-based partitioning paradigm, and discuss related work. For reference, Table 1 lists symbols and definitions that are used throughout the paper.

2.1 Problem definition

We consider data objects which are located in a metric space $\mathcal{D}$. Given two data objects r and $\bar{r}$, $|r, \bar{r}|$ represents the distance between r and $\bar{r}$ in $\mathcal{D}$. $|r, \bar{r}|$ is quantified by some similarity measurement which is a metric, i.e., it satisfies four requirements:

- $|r, \bar{r}| \geq 0$ (nonnegative);
- $|r, \bar{r}| = 0$ iff $r = \bar{r}$;
- $|r, \bar{r}| = |\bar{r}, r|$ (symmetry);
- $|r, \bar{r}| \leq |r, r'| + |r', \bar{r}|$ (triangle inequality).

Definition 1 (*Similarity Search*) Given a query object q, a data set R, and a threshold θ, the similarity search identifies objects in R with distance to q less than or equal to θ, i.e., $\{r | r \in R, |r, q| \leq \theta\}$.

Problem statement Unless otherwise specified, any object or data set we mention in the paper lies in the metric space $\mathcal{D}$. Without loss of generality, we assume there exists an M-attribute schema for data set R, in which the similarity is measured on colorbluea subset with N attributes (denoted as $\mathcal{A} : \{A_1, A_2 \ldots, A_N\})$ $N \leq M$. Moreover, R is maintained in a logically relational table with schema $\mathcal{R}$ before any queries are issued. In this paper, we focus on efficiently processing similarity queries in metric spaces using SQL statements.

2.2 Reference-based partitioning paradigm

Data partitioning Reference-based partitioning methods are widely used to process similarity queries in metric spaces, colorbluee.g., M-tree [9], iDistance [18] and PBI [23]. The rationale behind them is to first select m objects as pivots (pivots are highlighted in red in Fig. 2a), and then assign each object to a single pivot according to a certain strategy. For example, in iDistance, $\forall r \in R$ is assigned to its closest pivot (shown in Fig. 2a) or furthest pivot, while in PBI, $\forall r \in R$ is assigned to the pivot that maximizes the probability

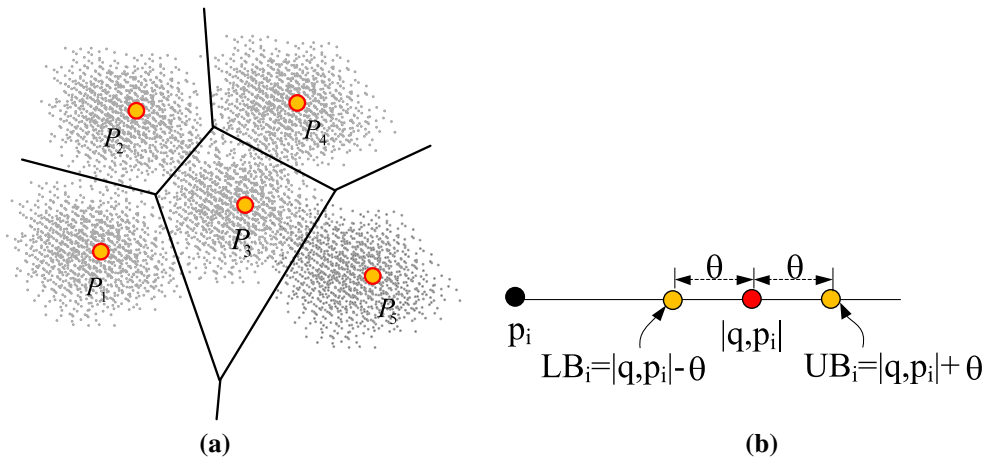

Fig. 2 An example of reference-based partitioning methods. **a** Assign objects to the closest pivots, **b** illustration of Theorem 1

Table 2 Pivot selection strategies

Approach	Domain	Pivot selection
iDistance [18,34]	Euclidean space	K-means
PBI [23]	Text space	MaxProb
M-Index [25]	Metric space	Random
MP-D [28]	Text space	MaxVar

of r being pruned with respect to a given query workload. After the assignment completes, the whole data space is split into m disjoint partitions. Let $\mathbb{P}$ be the set of pivots selected. $\forall p_i \in \mathbb{P}$, P_i^R denotes the partition with objects from R that take p_i as their pivot. Within each partition P_i^R, the distance $|r, p_i|$ from $\forall r \in P_i^R$ to p_i is also maintained.

Query processing When the whole data set is split into disjoint partitions, similarity queries can be efficiently processed by examining each partition P_i^R individually. Then, following the filtering-and-verification paradigm, the similar objects within P_i^R can be determined. The filtering rule is based on Theorem 1, in which $\forall r \in P_i^R$, $|r, p_i| \in [|p_i, q| - \theta, |p_i, q| + \theta]$ are taken as candidates. Finally, each of the candidates is verified and similar objects are returned. For ease of illustration, we refer to interval $[|p_i, q| - \theta, |p_i, q| + \theta]$ as the search range for partition P_i^R, and denote by LB_i and UB_i the lower and upper bound of the search range, respectively, i.e., $LB_i = |p_i, q| - \theta$ and $UB_i = |p_i, q| + \theta$ (Fig. 2b).

Theorem 1 [18,34] *Given a partition P_i^R, $\forall r \in P_i^R$, the necessary condition that $|q, r| \leq \theta$ is:*

$$LB_i = |p_i, q| - \theta \leq |p_i, r| \leq |p_i, q| + \theta = UB_i \tag{1}$$

Pivot selection Under Theorem 1, various pivot selection methods are proposed (listed in Table 2). **Random** used in M-Index randomly selects a set of objects from R as pivots. **k-means** used in iDistance partitions R into L clusters and the center of each cluster is taken as the pivots. **MaxProb** used in PBI selects a set of L pivots from R so that the expected number of candidates with respect to these pivots is minimized. **MaxVar** used in MP-D selects a set of L pivots from R so that the variance of objects in R with respect to these pivots is maximized.

2.3 Related work

Although there exists few work that processes similarity queries using SQL in more restricted Euclidean spaces and text spaces [15,31], to the best of our knowledge, this is the first solution in general metric spaces. Our study is related to previous work on **Native Solutions** and **SQL-based solutions** shown in Fig. 1.

Native solutions The majority of existing solutions belong to native solutions that focus on either (i) designing new index structures or (ii) designing new index probing algorithms with existing indices, as native engines to improve efficiency. Native solutions that design new index structures and new index probing algorithms to process efficient similarity queries include (1) the VP-Tree [32] and its variant [5], the GH-tree [27], GNAT [6], the M-Tree [9] and its variants [2,19], the D-Index [11], P-Sphere Tree [14], TV-Tree [22] over metric spaces, (2) kd-tree, Quadtree, Grid, X-tree over more restricted *Euclidean spaces*, and (3) Flamingo [4], the Tries [21] over more restricted *text spaces*. *Considering there do not exist any built-in indices that match these new index structures properly, it is hard to generalize and integrate such kind of solutions into existing RDBMS.*

The other native solutions adopt existing built-in index structures, like B^+-tree and R-tree, while developing new index probing algorithms. Among these solutions, the majority of them are designed for either the Euclidean spaces [18, 34] or the text spaces [23,36]. Although this kind of native solutions adopt built-in index structures used in existing RDBMS, these solutions require new index probing mechanisms which are not supported by existing RDBMS unless new APIs are implemented. Examples include the UB-tree, iDistance [18,34], and B^{ed}-tree [36], PBI [23], M-Index [25] and SPB-tree [8], respectively. For example, iDistance and PBI utilize a bi-directional probing algorithm, and M-Index adopts a recursive probing algorithm, requiring multiple-round traversals from the root to leaves, to retrieve candidates, while the original B^+-tree in RDBMS merely supports one-round, single-directional probing algorithm simultaneously. More importantly, as we shall discuss in Sect. 3, even when integrating this kind of solutions into RDBMS with newly introduced APIs, the performance of them could be degraded to table scans, nullifying the effect of index construction. Besides, these solutions were originally designed for static data sets and cannot work well in a dynamic workload that includes insert, update, and delete operations.

SQL-based solutions z-kNN [31] processes k nearest neighbor queries over *Euclidean spaces* in RDBMS using SQL. Specifically, by first efficiently extracting a set of approximate k nearest neighbors, it is able to obtain a tight search interval. z-kNN then issues a similarity query using this search interval to find similar objects, from which k objects with the smallest distances are finally identified as the k nearest neighbors. Text-SQL [15] is the only known method to process string similarity search under edit distance metric using SQL. Text-SQL determines the candidates by issuing a complex quad-table join. At the preprocessing phase, Text-SQL transforms each string r in R or q into a sequence of Q-grams, which are then maintained in two separate tables R' and and TQ'. At the query processing phase, Text-SQL issues a complex quad-table join on R, R', TQ and TQ' to determine all objects that share a certain number of Q-grams

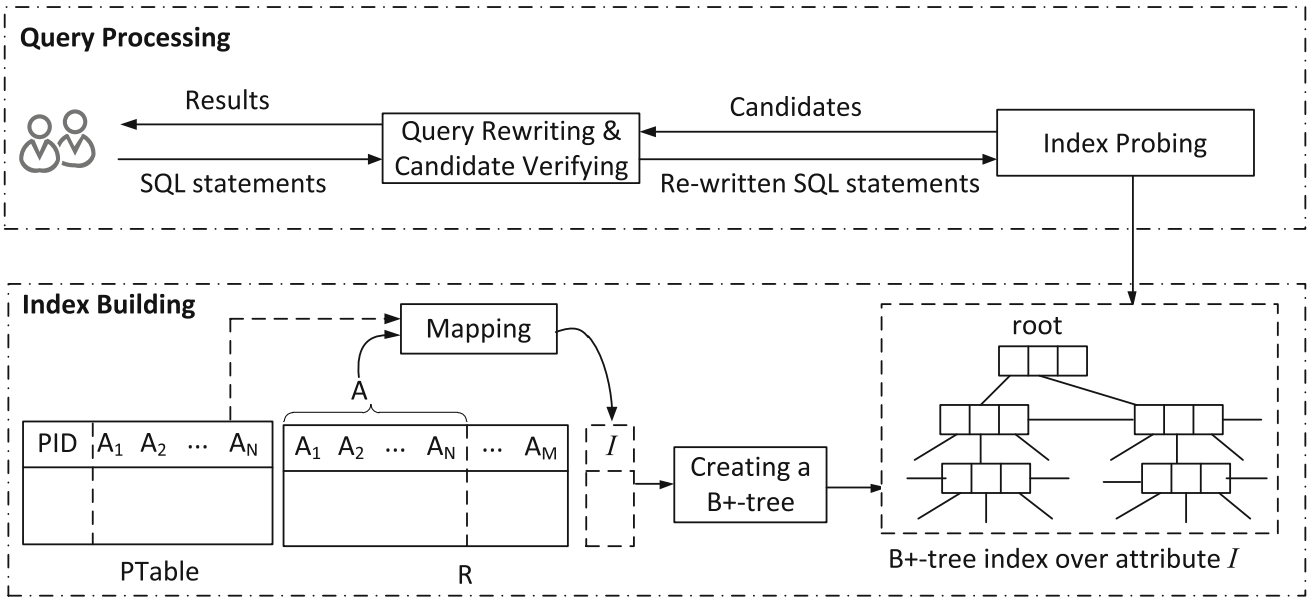

Fig. 3 An overview of MSQL

with q and selects them as candidates. Note that the cardinality of R' is multiple times larger than that of the input data set R. Obviously, performing the above quad-table join is rather expensive. Instead of conducting an expensive quad-table join, MSQL processes similarity queries by conducting an index join over R and another small relation, and hence improving the query performance significantly.

In this paper, we aim to propose a generic solution for similarity query processing in metric spaces that is independent on specific RDBMS. Hence, the proposed solution must be based on built-in indices that are fully supported in both commercial and open source databases. Such kind of indices includes B^+-tree, R-tree, and hash index, in which B+-tree is used for indexing pair-wise comparable objects, R-tree is used for indexing low-dimensional spatial data, and hash index is typically used for answering equivalent query. To the best of our knowledge, there do not exist any precise solutions that answer similarity queries in metric spaces based on the R-tree and hash index, while there exist some works showing that answering similarity queries using B^+-trees is feasible. Hence, in this paper, we propose a B^+-tree-based solution to process similarity queries in existing RDBMS.

3 Query processing in MSQL

3.1 An overview

Figure 3 gives an overview of MSQL, which processes the similarity query in the following two stages:

- *Index building* We utilize a mapping scheme that transforms values of objects over attributes $\mathcal{A}$ to pair-wise comparable signatures. Importantly, with this transformation, the index probing function used in the candidate selection phase is able to determine whether an object is definitely dissimilar to q by simply examining its signature. For this reason, we maintain signatures of objects in the same relation R by appending a new column named I and build a B^+-tree index over column I as all signatures are pair-wise comparable.
- *Query processing* To make the auxiliary information used in MSQL transparent to users, we provide a simple UDF SIMQ and users can obtain similar objects by issuing an SQL statement: SELECT SIMQ(R, $\mathcal{A}$, $q[\mathcal{A}]$, θ, $DIST$). We then rewrite the above SQL statement with the predicates that bound the signatures in a sequence of non-overlap, strict-increasing intervals to figure out candidates and the predicate DIST as a post-processing filter to verify the candidates. MSQL enables candidate selection using the pre-built B^+-tree index.

Our solution can be applied easily to any existing RDBMS that supports UDFs, such as PostgreSQL, MySQL, SQL Server, Oracle, etc.

3.2 Index building

In this subsection, we propose a signature generation scheme and present how to create the B^+-tree.

3.2.1 Signature generation scheme

To enable fast similarity query processing using index seeks, the generated signatures must satisfy the following requirements:

- Requirement 1: signatures are pair-wise comparable.
- Requirement 2: signatures located in a set of intervals are qualified as candidates.
- Requirement 3: let $S_i.min$ and $S_i.max$ be the minimum and maximum signatures of objects in partition P_i^R. Given two partitions P_i^R and P_j^R, there is no overlap between interval $[S_i.min, S_i.max]$ and interval $[S_j.min, S_j.max]$.

Requirement 1 makes sure that the signatures can be indexed using a B^+-tree, and Requirement 2 enables us to determine candidates by probing signatures in each interval using an index seek. Requirement 3 guarantees that signatures of objects in each partition are continuously maintained so that no signatures from other partitions will be accessed when traversing signatures of a partition.

With respect to existing reference-based partitioning methods, we find that most of them maintain all signatures in main memory and no indexing methods over signatures are considered. Three related works indexing signatures using B^+-trees are iDistance [18,34], iDistance variants M-Index [25] and PBI [23], which map each object $r \in P_i^R$ into a one-dimensional value ($V(r)$) based on the following equation:

$$V(r) = i * c + |r, p_i|$$

where c is a constant and must be set sufficiently large in order to avoid overlap between interval $[S_i.min, S_i.max]$ and interval $[S_j.min, S_j.max]$. In practice, c is difficult to set, since c largely depends on the application scenarios and data distributions. An improper c could result in a mixture of signature ranges from different partitions and further generate an incorrect set of candidates when processing similarity queries. To address this issue practically, we propose a signature generation scheme presented in Definition 2 using which the generated signatures completely satisfy the three requirements.

Definition 2 (*Signature Generation Scheme*) Given a dataset R and a set of pivots $\mathbb{P}$, $\forall r \in R$, we assign r to its closest pivot in $\mathbb{P}$. $\forall P_i^R \subseteq R$, $\forall r \in P_i^R$, we set the signature (denoted as $S(r)$) of r as $\langle i, |r, p_i|\rangle$, where i is the partition ID in R. □

Given two signatures $\langle i, d\rangle$ and $\langle j, d'\rangle$, we compare $\langle i, d\rangle$ and $\langle j, d'\rangle$ as follows:

$$\begin{cases} \langle i,d\rangle > \langle j,d'\rangle, & \text{if } i > j \text{ or } (i = j \text{ and } d > d'), \\ \langle i,d\rangle = \langle j,d'\rangle, & \text{if } i = j \text{ and } d = d', \\ \langle i,d\rangle < \langle j,d'\rangle, & \text{otherwise;} \end{cases} \quad (2)$$

Theorem 2 *Signatures generated using Definition 2 meet the above three requirements.* □

Proof We shall prove that our signature generation scheme meets the three necessary requirements separately.

- Requirement 1: signatures are pair-wise comparable. This requirement could be directly proved based on our comparison rule.
- Requirement 2: signatures locating in a set of intervals are qualified as candidates. We consider each partition separately. For each partition P_i, based on Theorem 1, only objects with distances to p_i in range $[|p_i, q| - \theta, |p_i, q| + \theta]$ are taken as candidates, while in our cases, the signatures for these objects reside in $[\langle i, |p_i, q| - \theta\rangle, \langle i, |p_i, q| + \theta\rangle]$.
- Requirement 3: Given two partitions P_i^R and P_j^R, there is no overlap between interval $[S_i.min, S_i.max]$ and interval $[S_j.min, S_j.max]$. Given a partition P_i^R, the minimum and maximum signatures $S_i.min$ and $S_i.max$ of objects in P_i^R are $\langle i, 0\rangle$ and $\langle i, \infty\rangle$. Similarly, given another partition P_j^R, $S_j.min$ and $S_j.max$ of objects in P_j^R are $\langle j, 0\rangle$ and $\langle j, \infty\rangle$. According to our comparison rule, $\langle i, 0\rangle > \langle j, \infty\rangle$ if $i > j$ and $\langle j, 0\rangle > \langle i, \infty\rangle$ otherwise (Note $i \neq j$). Therefore, there is no overlap between intervals $[S_i.min, S_i.max]$ and $[S_j.min, S_j.max]$. □

The signature generation scheme is able to keep the partitions non-intersect (Requirement 3). As a result, when probing signatures of candidates in an individual partition, the probing scheme is able to continuously access the corresponding index pages, while no signatures from the other partitions in the remaining index pages are probed, hence avoiding unnecessary I/O and redundant computations.

3.2.2 B^+-tree construction

Based on the signature generation scheme and the comparison rule, we can now build the index over the signatures as follows:

- *Storing pivots in PTable* We suppose the pivots have been selected and stored in a table called **PTable**.[1] PTable consists of attribute PID (pivot ID) with integer data type and its original attributes $\mathcal{A}$;
- *Appending the attribute I to R* We append a new column I to R, and $\forall r \in P_i^R$, we update $r[I]$ (i.e.,$S(r)$) to $\langle i, |r, p_i|\rangle$.
- *Building the B^+-tree index* As a composite data type, we need to integrate the comparison rule as a UDF and then build an B^+-tree index over attribute I. An alternative solution is to decompose I into two columns I_1, I_2, and

[1] We will propose the pivot selection method in Sect. 4.2.

build a composite index over (I_1, I_2). The following SQL statements show the details on how to build the index.

```
1: CREATE TYPE iPair as (key INTEGER,value DOUBLE);
2: ALTER TABLE R ADD I iPair;
3: UPDATE R set R.I = (
       SELECT (S.PID, DIST(S[A], R[A]))::iPair as I1
       FROM PTable S
       ORDER BY I1.value
       LIMIT 1);
4: CREATE INDEX I-INDEX ON R
       USING BTREE (I ASC NULL LAST);
```

Example 1 (Index Building) Suppose there is a relation R with three attributes: **ID** (customer ID), **Name** (customer names), and **Coord** (customer location coordinates). R has 10 records (see Fig. 4). To speed up nearby location query processing, before any query is issued, we first build a PTable with two attributes PID and Coord. Suppose we have selected 5 pivots stored in PTable. Subsequently, we transform R to R' by appending a signature attribute I_coord, in which $\forall r \in R$, we update $S(r)$ to $\langle i, |r, p_i| \rangle$ based on Definition 2. Transformation from R to R' is shown in the right part of Fig. 4 ($L1$ is used as the similarity metric). Finally, we build the B^+-tree index over attribute I_coord. *Note that similar index construction can be made over* **Name** *attribute if we target to efficiently answer approximate customer name queries. We omit the details as they follow the same index building routine but with different similarity measures.* □

3.3 Data manipulation

MSQL fully utilizes RDBMS-based techniques to handle manipulation, including data insertion, data update, and data deletion.

- *Data insert and update* We create an **after insert, update trigger** on input relation R. After an insert or update operation of any object r is executed, the trigger is fired to invoke the following steps: (1) finding the closest pivot residing in PTable to r, (2) computing the signature of r based on its closest pivot, and (3) updating I attribute value using this signature accordingly. As we build a B^+-tree index over attribute I, the index will be automatically maintained by RDBMS. To do data insert and update, the averaged time cost of computing the signature of r is $|\mathbb{P}| * \Delta$, where Δ is the averaged distance computation cost in R. The time complexity of updating the B^+-tree index is $O(\log |R|)$. In practice, computing the signature of each object could be accelerated by existing indexing techniques, but this optimization is out the scope of this paper.
- *Data deletion* As we maintain signatures in a separate column of the same relation, for an object r to be deleted, the signature of r will be removed as well. Besides, the index will also be automatically updated by RDBMS. Therefore, we do nothing for data deletion. The time complexity of removing r from the original relation together with updating the B^+-tree index is $O(\log |R|)$.

3.4 Query processing

3.4.1 Baseline approach

As presented in Sect. 2.3, reference-based approaches, e.g., iDistance and its variants, process similarity queries using various probing mechanisms which are not directly supported in existing RDBMS. *For this reason, we abstract the core techniques of reference-based approaches using SQL to process similarity queries.* The rationale behind the reference-based approaches is to follow the filtering-and-verification paradigm, i.e., (1) figure out candidates by examining objects in each partition separately based on Theorem 1 and (2) verify candidates by computing their distances to the query.

Lemma 1 *Given a data set R, $\forall r \in R$ with signature $S(r) = \langle i, |p_i, r| \rangle$, the necessary condition that $|q, r| \leq \theta$ is:*

$$LB_i \leq S(r) \leq UB_i \tag{3}$$

where $LB_i = \max\{\langle i, |p_i, q| - \theta \rangle, \langle i, S_i.min \rangle\}$ and $UB_i = \min\{\langle i, |p_i, q| + \theta \rangle, \langle i, S_i.max \rangle\}$.[2]

Proof First, based on Definition 2, object r with signature $S(r) = \langle i, |p_i, r| \rangle$ belongs to partition P_i^R. Based on Theorem 1, the necessary condition that $|q, r| \leq \theta$ is $|p_i, q| - \theta \leq |p_i, r| \leq |p_i, q| + \theta$, i.e., $\langle i, |p_i, q| - \theta \rangle \leq S(r) \leq \langle i, |p_i, q| + \theta \rangle$. Second, $S_i.min$, $S_i.max$ are the minimum and maximum distances of objects residing in P_i^R to p_i. As $r \in P_i^R$, we have $\langle i, S_i.min \rangle \leq S(r) \leq \langle i, S_i.max \rangle$. Therefore, we can derive Eq. 3. □

Based on Lemma 1, it is able to answer similarity queries by issuing an SQL statement below:

```
SELECT   R.A1, ..., R.AN
FROM     R
WHERE    LOCATE(R.I, @ranges) AND
         DIST(R[A], q[A], θ);
```

[2] As pointed out by existing reference-based approaches, by explicitly maintaining $S_i.min$ and $S_i.max$, the whole partition can be pruned when $|p_i, q| - \theta > S_i.max$, i.e., $UB_i < LB_i$.

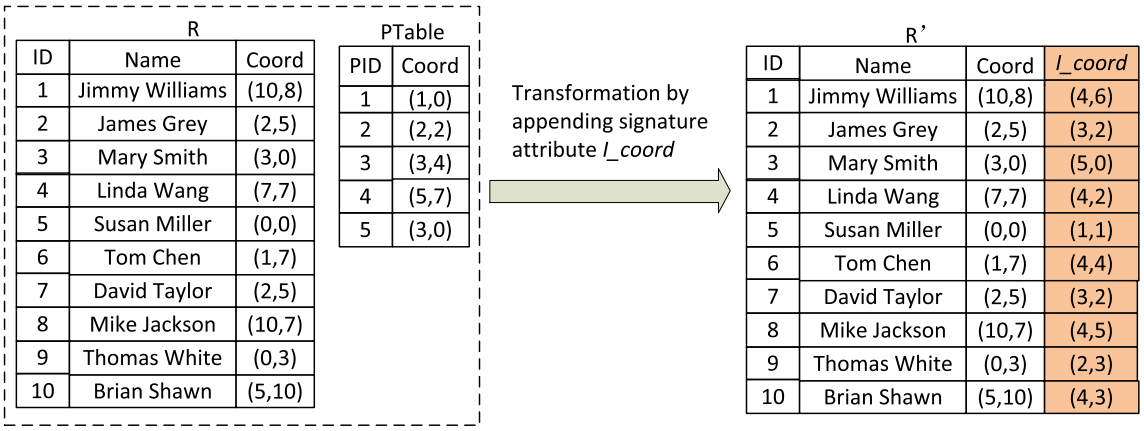

Fig. 4 An example of building the B+-tree index

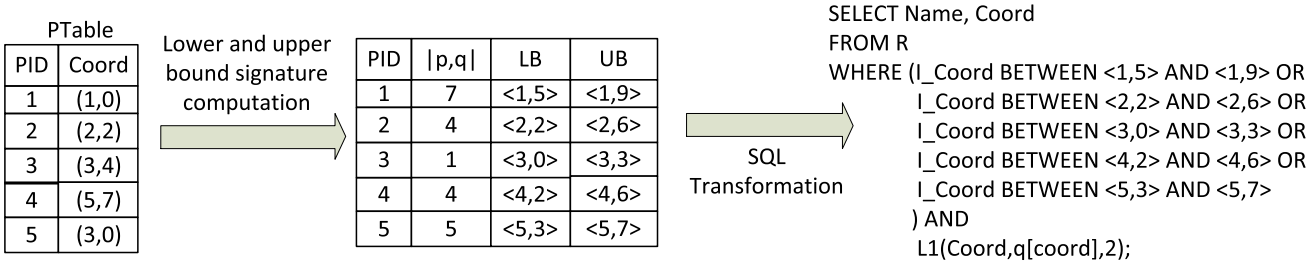

Fig. 5 A running example of the baseline approach with query $q[coord] = (4, 4)$ and $\theta = 3$ under $L1$ similarity metric

In the above SQL statement, predicate LOCATE($R.I$, @ranges) bounds the signatures, $R.I$, in a collection of search ranges, denoted as @ranges=$\cup_{i\in[1,|\mathbb{P}|]}\langle LB_i, UB_i\rangle$, to figure out candidates, and predicate DIST as a post-processing filter to verify the candidates. Hence, a full SQL statement to process similarity search by unfolding the predicate LOCATE($R.I$, @ranges) is shown below:

```
SELECT   R.A_1, ..., R.A_N
FROM     R
WHERE    (R.I between LB_1 and UB_1        OR
          R.I between LB_2 and UB_2        OR
          ...                              OR
          R.I between LB_|P| and UB_|P|)   AND
          DIST(R[A], q[A], θ);
```

Example 2 (Baseline Approach) Continue the example that is presented in Fig. 4. Suppose that a user issues a query to find customers with nearby locations, e.g., close to coordinate $q[coord] = (4, 4)$ within $L1$ distance 2. Figure 5 gives a running example of how the baseline approach works. Specifically, we compute $|p_i, q|$, LB_i and UB_i, $\forall p_i$ in PTable, and then issue an SQL statement (shown on the right part of Fig. 5) to find nearby customers. □

Typically, each search range condition triggers an index seek on the composite index and a progressive heap scan is conducted to merge the candidates from individual index seeks. Nevertheless, as we observe in many cases[3] (e.g., when the size of the data set is large or the dimensionality of the data set is relatively high), to achieve a good query performance, it is necessary to select thousands or even ten thousands of pivots, leading to a large number of search range conditions. As we will discuss later, using an increasing number of search range conditions could result in a smaller number of candidates. Nevertheless, using a large number of search range conditions potentially causes high computation cost. The reason is twofold. First, it significantly increases the cost of selecting the best query plan in the query optimizer. For example, according to our experiments, the optimizer in PostgreSQL even takes 2ms to generate the query plan in the SQL statement involving in only 10 search range conditions. *Second and worse still, it resorts to table scan instead of index seek when the number of search range conditions exceeds a threshold, nullifying the effect of index seek*. To address these two issues in practice, we then propose a join-based query rewriting scheme.

3.4.2 Join-based approach

To make the implementation details transparent to users, i.e., it is unnecessary for users to be familiar with the

[3] This observation is also investigated in [23].

Table 3 An example of SearchRangeSet

LB	UB
$\langle 1, 5\rangle$	$\langle 1, 9\rangle$
$\langle 2, 2\rangle$	$\langle 2, 6\rangle$
$\langle 3, 0\rangle$	$\langle 3, 3\rangle$
$\langle 4, 2\rangle$	$\langle 4, 6\rangle$
$\langle 5, 4\rangle$	$\langle 5, 7\rangle$

implementation details of processing similarity queries in MSQL, we provide a simple UDF SIMQ so that users can obtain similar objects by issuing an SQL statement: SELECT SIMQ($R, \mathcal{A}, q[\mathcal{A}], \theta, DIST$). After receiving the above SQL statement, we first build a temporary schema, named as `SearchRangeSet`, which stores the lower bound and upper bound of every search range with respect to a given query. We then compute $[LB_i, UB_i]$ for each partition P_i^R, and insert $[LB_i, UB_i]$ ($i \in [1, |\mathbb{P}|]$) into `SearchRangeSet`, an example of which is listed in Table 3, where the query and the pivots are given under the settings shown in Fig. 5. We finally rewrite the SELECT SIMQ($R, \mathcal{A}, q[\mathcal{A}], \theta, DIST$) statement as the following join-based SQL statement.

```
SELECT   R.A_1, ..., R.A_N
FROM     R, SearchRangeSet SRS
WHERE    I BETWEEN SRS.LB and SRS.UB AND
         DIST(R[A], q[A], θ);
```

In the above SQL statement, we perform a Cartesian product over two relations R, SearchRangeSet with the predicate that qualifies records in R with signatures $R.I$ residing in any range of $\langle SRS.LB, SRS.UB\rangle$. As the cardinality of relation SearchRangeSet is relatively small and signatures of R are built with an index, the execution engine will resort to an index join to retrieve the candidates and similar objects are returned by verifying the candidates with the filter DIST.

3.4.3 Discussion

Extension for kNN query processing MSQL could potentially be extended to support k nearest neighbor (a.b.a kNN) search that returns k objects from R with the smallest distances to q. Like existing reference-based approaches, we can transform kNN search into a sequence of similarity searches, each of which is processed using MSQL. To avoid redundant computation, we make two slight adjustment of MSQL. First, we use DIST($R[\mathcal{A}], q[\mathcal{A}], \theta$) as the **computation** function rather than the **verification** function. Second, search ranges of each partition that have been examined in the previous iteration will be removed in the current iteration. Take two continuous similarity queries with threshold θ and θ', respectively, for example. According to Theorem 1, the search range with threshold θ' in partition P_i^R is $[|p_i, q| - \theta', |p_i, q| + \theta']$. However, because objects with distances to p_i in $[|p_i, q| - \theta, |p_i, q| + \theta]$ have been examined in the previous iterations. Thus, we adjust the search range in the current iteration to $[|p_i, q| - \theta', |p_i, q| - \theta) \cup (|p_i, q| + \theta, |p_i, q| + \theta']$. In this way, no objects will be examined twice. In this paper, we focus on the similarity query processing and leave the kNN query processing as our future work.

Limitations of the join-based approach Although the join-based approach avoids issuing a table scan over R, there are still two issues that are required to address.

- To obtain LB_i and UB_i in Eq. 3, it is necessary to maintain $S_i.min$ and $S_i.max$ in **PTable** for every partition. However, introducing $S_i.min$ and $S_i.max$ will cause extra maintenance cost. For any delete, insert, update of an object $r \in P_i^R$, we need to check whether $S_i.min$ and $S_i.max$ are required to update. For any insert of an object $r \in P_i^R$, we first need to examine whether $|p_i, r| = S_i.min$ or $|p_i, r| = S_i.max$ and if so, we then replace $S_i.min$ or $S_i.max$ using $|p_i, r|$ accordingly. The examination cost takes the time complexity $O(\log |\mathbb{P}|)$. For any delete of an object $r \in P_i^R$, we examine whether $|p_i, r| = S_i.min$ or $|p_i, r| = S_i.max$, and if so, we update $S_i.min = \min_{\forall o \in P_i^R \wedge o \neq r} |p_i, o|$ or $S_i.max = \max_{\forall o \in P_i^R \wedge o \neq r} |p_i, o|$. The time complexity of this update is $O(\log |R|)$. To update object r with r', we need to examine whether any of $|p_i, r|$ and p_i, r' equals to $S_i.min$ or $S_i.max$, and if so, we need to update $S_i.min$ and $S_i.max$ accordingly. The time complexity of this update is $O(\log |R|)$ as well. Apparently, in a dynamic workload environment, any delete, insert, update operation will cause a conflict with the select operation in **PTable**, and hence degrading the similarity query performance. To verify this observation, we conduct a comprehensive experiment study in Sect. 5.2.3 on how the performance of processing the similarity queries is affected when a certain number of delete, insert, update operations occur simultaneously.
- As pointed out in [23,24], hundreds or thousands of pivots are often used in order to speed up similarity query processing. Although it is able to generate less number of candidates by using an increasing number of pivots, in many cases, the bound of each search range $[LB_i, UB_i]$ is still loose and the majority of partitions still cannot be entirely pruned. In this way, it is arguably hard to guarantee the efficiency and scalability of the proposed SQL-based solution since a large number of index seeks are still invoked.

To address the above two issues, it is necessary to propose a new pruning rule that obtain a tighter bound of $[LB_i, UB_i]$ without maintaining $S_i.min$ and $S_i.max$.

4 Query optimization

By bounding the search range $[LB_i, UB_i]$ in partition P_i^R, only the objects whose signatures locate in $[LB_i, UB_i]$ are considered as candidates. Therefore, to minimize the number of candidates, we need to find a tight $[LB_i, UB_i]$. In this section, we introduce two optimizations to achieve a near optimal range selection:

- A new pruning rule: Compared with Theorem 1, the new pruning rule can reduce the upper bound of search range for each partition P_i^R, while keeping the lower bound unchanged, thereby enhancing the pruning power. Due to this new pruning rule, we do not necessarily maintain $S_i.min$ and $S_i.max$ and hence avoid extra maintenance cost.
- A pivot selection mechanism: Based on our signature generation scheme described in Definition 2, once the set of pivots is given, signatures of objects are deterministic. Therefore, the problem of minimizing the number of candidates is reduced to how to select a proper set of pivots so that the number of objects located with their signatures in $\bigcup_{i=1}^{|\mathbb{P}|}[LB_i, UB_i]$ is minimized. We prove that selecting the optimal set of pivots is NP-hard, and hence we propose an efficient heuristic approach instead.

4.1 Enhancing the filtering power with a new pruning rule

By Theorem 1, $\forall r \in P_i^R$, r is taken as a candidate as long as $|p_i, r| \in [|p_i, q| - \theta, |p_i, q| + \theta]$. The above pruning rule filters out dissimilar objects in partition P_i^R merely based on p_i while omitting the other pivots. Observing that in our data partitioning scheme we assign each object to its closest pivot, we consider whether it is possible to further filter out more dissimilar objects using the other pivots.

Example 3 (Motivating Example) Continue the examples presented in Figs. 4 and 5. Under Theorem 1, candidates in partition P_4^R are $r_1, r_4, r_6, r_8, r_{10}$ since their signatures lie in range $[LB_4, UB_4]$. Consider the candidate r_1 ($r_1[Coord] = (10, 8)$) with $L1$ distance = 6 to its closest pivot p_4 ($p_4[Coord] = (5, 7)$). Remind that each object is assigned to its closest pivot. Hence, $\forall p \in \mathbb{P}$, we can have $|r_1, p| \geq 6$ without computing $|r_1, p|$. Now, consider pivot p_3. A record is taken as a candidate regarding p_3 only if its signature is bounded in range $[LB_3, UB_3]$, i.e., $[\langle 3, 0\rangle, \langle 3, 3\rangle]$. Apparently, regarding p_3, r_1 is verified to be dissimilar to q since $\langle 3, |r_1, p_3|\rangle \notin [LB_3, UB_3]$. Similarly, r_6 and r_8, underlined in Table 4, can also be pruned based on p_3. □

As discussed in Example 3, with respect to partition P_i^R, we can further filter out some objects that were originally taken as candidates in P_i^R using the other pivots. This leads to another question we need to answer: *which candidates in P_i^R can be pruned safely?*. To answer this question, we consider the following case:

Table 4 Illustration of the new pruning rule

Record	$\lvert p_4, r\rvert$	$S(r)$
r_1	6	$\langle 4, 6\rangle$
r_4	2	$\langle 4, 2\rangle$
r_6	4	$\langle 4, 4\rangle$
r_8	5	$\langle 4, 5\rangle$
r_{10}	3	$\langle 4, 3\rangle$

- $\exists p_i, p_j \in \mathbb{P}, |p_j, q| + \theta < |p_i, q| - \theta$. In this case, it is guaranteed that for any candidate $r \in P_i^R$, i.e., $|p_i, q| - \theta \leq |r, p_i| \leq |p_i, q| + \theta$, we have $|p_j, q| + \theta < |p_i, q| - \theta \leq |r, p_i| \leq |r, p_j|$ and hence, all objects in partition P_i^R can be completely pruned.

Based on this observation, we split the interval $[|p_i, q| - \theta, |p_i, q| + \theta]$ into two intervals $[|p_i, q| - \theta, |p_j, q| + \theta]$ and $(|p_j, q| + \theta, |p_i, q| + \theta]$, if $|p_j, q| + \theta \in [|p_i, q| - \theta, |p_i, q| + \theta]$. Because a candidate r is pruned if $|r, p_i|$ locates in the second interval, we only examine the remaining objects, i.e., those candidates with $|r, p_i|$ locating in the first interval $[|p_i, q| - \theta, |p_j, q| + \theta]$. This pruning rule is always true for any other pivot p_j in $\mathbb{P}$. Hence, we have the following theorem:

Theorem 3 *Given a query q, let p_q be the pivot in $\mathbb{P}$ with the minimum distance to q. Formally,*

$$p_q = \arg\min_{p \in \mathcal{P}} |q, p| \tag{4}$$

Given a partition P_i^R, $\forall r \in P_i^R$, the necessary condition that $|q, r| \leq \theta$ is:

$$|p_i, q| - \theta \leq |p_i, r| \leq |p_q, q| + \theta \tag{5}$$

Proof Given a partition $P_i^R, \forall r \in P_i^R$, according to Theorem 1, the necessary condition that $|q, r| \leq \theta$ is: $|p_i, q| - \theta \leq |p_i, r| \leq |p_i, q| + \theta$. As $|p_q, q| \leq |p_i, q|$, we need to prove that objects belonging to P_i^R with distances to q in the range $(|p_q, q| + \theta, |p_i, q| + \theta]$ are definitely dissimilar to q.

We provide the proof by contradiction. Suppose there exists an object $r \in P_i^R$ with distance to q in $(|p_q, q| + \theta, |p_i, q| + \theta]$ and r is similar to q. According to the triangle inequality, if $|r, q| \leq \theta$, then $\forall p_j \in \mathbb{P}$, we have $|p_j, q| - \theta \leq |r, q| \leq |p_j, q| + \theta$. Therefore, considering p_q is one pivot in $\mathbb{P}$, we can have $|r, q| \leq |p_q, q| + \theta$. According to our partitioning method, each object in R is assigned to the partition with the nearest pivot, and hence we can have $|r, p_i| \leq |r, p_q|$. This contradicts our assumption that $|r, p_i| > |p_q, q| + \theta > |r, p_i| + \theta$. □

Compared with Theorem 1, for each search range, Theorem 3 is able to derive a tighter upper bound while keeping the lower bound unchanged. For this reason, Theorem 3 can get a tighter window size for each search range. Further, as long as we collect sufficiently many good-quality pivots $\mathbb{P}$, for any query q, if there exist a pivot $p_q \in \mathbb{P}$ that produces a small $|p_q, q|$, we can skip examining the entire objects in each partition P_i^R with $|p_i, q| - \theta > |p_q, q| + \theta$, and hence reduce both CPU and I/O cost significantly. For this reason, we do not explicitly maintain $S_i.min$ and $S_i.max$ in PTable, and hence we are able to avoid extra maintenance cost and locking overhead over PTable.

To answer similarity queries under the new pruning rule, we do not need to modify any SQL statements. Instead, we update UB_i and LB_i in Lemma 1, which are set to $\langle i, |p_q, q| + \theta \rangle$ and $\langle i, |p_i, q| - \theta \rangle$, respectively. In the remainder of this paper, unless otherwise specified, UB_i and LB_i are set to $\langle i, |p_q, q| + \theta \rangle$ and $\langle i, |p_i, q| - \theta \rangle$, respectively.

4.2 Pivot selection

The number of candidates depends on the set of search ranges $\bigcup_{i=1}^{|\mathbb{P}|}[LB_i, UB_i]$ as well as the signatures of objects. Considering that search ranges and signatures both depend on the selected pivots, it is clear that the problem of selecting pivots is crucial. Specifically, we want to select a given number of L pivots so that the candidate size is minimized. L is a tuneable parameter.

4.2.1 Cost model

Given a pivot $p_i \in \mathbb{P}$, let $f_{p_i}(x, Q)$ be the probability density function (PDF) that describes the relative likelihood for random variable x to take on a given value with respect to pivot p_i over the query load Q. Formally, $f_{p_i}(x, Q)$ is defined as:

$$f_{p_i}(x, Q) = \frac{|\{q | q \in Q, |q, p_i| = x\}|}{|Q|}. \tag{6}$$

Suppose we have selected a set of pivots $\mathbb{P}$ to do partitioning. Our cost model is motivated by the following question: *given an object r, a set of pivots $\mathbb{P}$, and a query load Q, how many times is r taken as a candidate over Q with respect to $\mathbb{P}$?* Indeed, this problem can be generalized as: *given an object r and a set of pivots $\mathbb{P}$, how much is the probability (denoted as $\mathcal{P}_{\mathbb{P}}(r)$) of r taken as a candidate when answering similarity queries with respect to $\mathbb{P}$?* As r is merely maintained in partition P_i^R, we can have $\mathcal{P}_{\mathbb{P}}(r) = P_{\{p_i\}}(r)$.

To answer the above questions, we first give an example of discrete probability density function $f_{p_i}(x, Q)$ in Fig. 6. Suppose there exists an object r (shown in red color) and

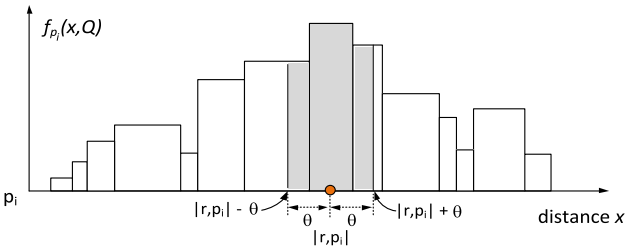

Fig. 6 A necessary condition to select r as a candidate

r is assigned to p_i.[4] Regarding p_i, once a query q locating outside the interval $[|p_i, r| - \theta, |p_i, r| + \theta]$, we can guarantee that $|r, q| > \theta$, i.e., only for $\forall q \in Q$ with $|p_i, q| \in [|p_i, r| - \theta, |p_i, r| + \theta]$, r is possibly taken as a candidate. Further, as discussed in Theorem 3, to finally select r as a candidate, there cannot exist any other pivot $p_j \in \mathbb{P}$ such that $|p_j, q| + \theta \leq |p_i, r|$, i.e., among these $q \in Q$ with $|p_i, q| \in [|p_i, r| - \theta, |p_i, r| + \theta]$, r is finally qualified as a candidate only if $|p_q, q| + \theta \geq |p_i, r|$. Therefore, we classify the relationship among q, $\mathbb{P}$ and r into two categories:

- **COND1:** $|p_i, q| \in [|p_i, r| - \theta, |p_i, r| + \theta]$
- **COND2:** $|p_i, q| \in [|p_i, r| - \theta, |p_i, r| + \theta]$ and $|p_q, q| + \theta \geq |p_i, r|$.

Let $\phi_{p_i}(r)$ be the probability that condition COND1 is satisfied, and let $\phi'_{p_i}(r)$ be the probability that condition COND2 is satisfied. Theoretically, $P_{p_i}(r)$ is the conditional probability measuring the probability $|p_q, q| + \theta \geq |p_i, r|$ under the condition $|p_i, q| \in [|p_i, r| - \theta, |p_i, r| + \theta]$, and hence $P_{p_i}(r)$ can be formalized as:

$$\mathcal{P}_{\mathbb{P}}(r) = P_{\{p_i\}}(r) = \frac{\phi'_{p_i}(r)}{\phi_{p_i}(r)}, \tag{7}$$

where $\phi_{p_i}(r)$ and $\phi'_{p_i}(r)$ are

$$\phi_{p_i}(r) = \sum_{|p_i,q| \in [|p_i,r|-\theta, |p_i,r|+\theta]} f_{p_i}(x, Q), \tag{8}$$

$$\phi'_{p_i}(r)n = \sum_{|p_i,q| \in [|p_i,r|-\theta, |p_i,r|+\theta] \wedge |p_q,q|+\theta \geq |p_i,r|} f_{p_i}(x, Q). \tag{9}$$

Given a set of pivots $\mathbb{P}$, to answer similarity queries, we can now compute the expected number of candidate objects in R with regard to $\mathbb{P}$ as follows:

Lemma 2 *Given a set of pivots $\mathbb{P}$ and a data set R, to answer a similarity query with threshold θ, the expected number $E_{\mathbb{P}}(R)$ of objects in R to be regarded as candidates is*

$$E_{\mathbb{P}}(R) = \sum_{P_i^R \subseteq R} \sum_{r \in P_i^R} P_{p_i}(r). \tag{10}$$

[4] Unless otherwise specified, we assume that r is assigned to pivot p_i in the remainder of this paper.

Definition 3 *(Pivot Selection)* Given a query load Q, a data set R, the problem of pivot selection is to select L pivots, $\mathbb{P}$, so that $E_{\mathbb{P}}(R)$ defined in Eq. (10) is minimized:

$$\arg\min_{\mathbb{P}} \sum_{P_i^R \subseteq R} \sum_{r \in P_i^R} P_{p_i}(r). \tag{11}$$

Theorem 4 *Finding the optimal solution to pivot selection is NP-hard.*

Proof The k-means clustering problem is NP-hard when the number of dimensions is not <2 [10]. We then reduce the k-means clustering problem to the optimal pivot selection problem. Equation 10 quantifies the expected number $E_{\mathbb{P}}(R)$ of objects in R to be regarded as candidates, in which $E_{\mathbb{P}}(R)$ is computed by aggregating the probability of r taken as a candidate regarding $\mathbb{P}$, $\forall r \in R$. The optimal pivot selection problem is essentially the k-means problem with a different optimization objective, i.e., the latter targets to minimize the sum of the distance from each point to its closest cluster center, while the former targets to minimize the sum of the probability for each object being as a candidate regarding a set of pivots. k-means clustering problem is merely applied in Euclidean spaces, while the optimal pivot selection problem is applied in more general metric spaces with more complex optimization objective function. Hence, we can reduce the k-means clustering problem to our problem. □

4.2.2 A heuristic approach to pivot selection

According to Theorem 4, answering the optimal pivot selection problem is NP-hard. We therefore propose a heuristic approach to extracting the pivots greedily.

Before proceeding to present our heuristic approach, we want to clarify that, although there exist some existing approaches which aim to select good pivots, they cannot be directly applied to solve our problem. The reasons are twofold. First, some of them are particularly designed for restricted Euclidean spaces [18,34] and cannot be extended for general metric spaces. For example, k-means used in iDistance works merely in Euclidean spaces. Second and more importantly, all existing approaches select the pivots under Theorem 1. In this way, objects that are distant to others are often selected as pivots [18,23,28,34]. In this paper, we apply Theorem 3 instead of Theorem 1 as the pruning rule. Theoretically, we aim to obtain a minimum $|p_q, q|$ so that UB_i for each partition can be minimized. That is, we aim to select the pivots $\mathbb{P}$ so that $\forall r \in R, \exists p \in \mathbb{P}$, r is close to p. Due to the above two reasons, we propose a new heuristic approach to selecting pivots.

In order to serve as a generic solution that is independent of a specific application domain, we make the following assumption:

– $\mathbb{P} \subset (R \cup Q)$.

Our pivot selection approach is motivated by the following observation:

Observation 1 *Given a query q, if there exists a pivot $p_i \in \mathbb{P}$ with $q = p_i$, then the upper bound UB_j of the search range for each partition P_j^R is minimized, which is $\langle j, \theta \rangle$.*

Example 4 Consider data set R is a collection of strings and edit distance is used as the similarity function. Suppose we have select three pivots: "Robert Morris," "James Dean," and "Michael Morris." Let query q and threshold θ be "Robert Morris" and 1, respectively. First, we compute the distance between each pivot and q. In this case, we have $|p_1, q| = 0$, $|p_2, q| = 11$, and $|p_3, q| = 7$. We then obtain the search range for each partition, which is $[\langle 1, 0\rangle, \langle 1, 1\rangle]$ for P_1^R, $[\langle 2, 10\rangle, \langle 2, 1\rangle]$ for P_2^R, and $[\langle 3, 6\rangle, \langle 3, 1\rangle]$ for P_3^R. As the latter two range searches are invalid, we then only set the range search $[\langle 1, 0\rangle, \langle 1, 1\rangle]$ in the predicate. □

Observation 1 inspires us to select the pivots so that $\forall q \in Q$, there exists a pivot which is exactly the same as q. In this way, the upper bound of the search range for each partition P_i^R is minimized, making the maximum possibility for P_i^R be entirely pruned. Nevertheless, in many cases, like in high-dimensional spaces, finding such a set of pivots is practically impossible. Instead, we expect to find a set of pivots $\mathbb{P} \subset (R \cup Q)$ so that $\forall q \in Q$, there exists a pivot p_i, making the distance $|q, p_i|$ as small as possible. For this reason, we relax our problem statement of the pivot selection below:

Definition 4 (*Approximate Pivot Selection*) Given a query load Q, a data set R, the problem of approximate pivot selection is to select L pivots, $\mathbb{P} \subset (R \cup Q)$, so that the sum of distances from $\forall q \in Q$ to its closest pivot in $\mathbb{P}$ is minimized.

Theoretically, k-medoids [20] is the best solution to solve the approximate pivot selection problem. Generally, applying k-medoids to select pivots can be described as follows. At step 1, we select L objects from $(R \cup Q)$ as $\mathbb{P}$. At step 2, $\forall q \in Q$, we then assign q to its closest pivot in $\mathbb{P}$. After the assignment completes, we compute the sum of distances from each q to its closest pivot in $\mathbb{P}$. At step 3, $\forall p \in \mathbb{P}, \forall p' \in (R \cup Q) - \mathbb{P}$, we swap p and p', do the reassignment of each $q \in Q$, and recompute the above sum. If the sum increases, we then undo swap. At step 4, we repeat executing Step 3 until the sum does not decrease. Following the above steps to select the pivots, the time complexity for each iteration (step 3) is $O(L \times (|R| + |Q| - L) \times |Q|)$ [20]. Apparently, when the size of Q or R is large, applying k-medoids to select pivots is prohibitively expensive. Considering that the size of Q and R could be large, we consequently propose a bipartite-based approach that selects pivots with only one iteration.

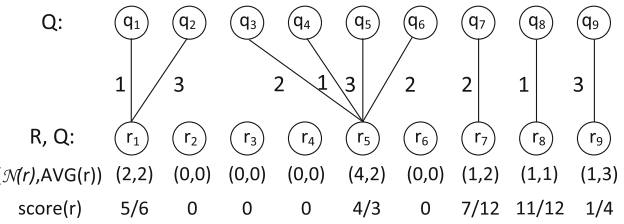

Fig. 7 Overview of the one-pass bipartite-based approach

In essence, our approach assigns a score to each object in $(R \cup Q)$ and L objects with greatest scores are selected as the pivots $\mathbb{P}$. To select a set of effective pivots, the score should be encoded properly so that $\forall q \in Q$, there exists a pivot p_i, making the distance $|q, p_i|$ as small as possible. Figure 7 shows the rationale behind our approach. We take Q, and $Q \cup R$ as two disjoint sets of vertices in a bipartite graph. $\forall q \in Q$, we find its closest object r in $(Q \cup R)$ and create an edge from q to r, denoted as $q \rightarrow r$. Since r is from set $(Q \cup R)$, there must exist an object $r = q$ taking the closest object to q, making the final pivot set $\mathbb{P}$ entirely from Q. To cope with this problem, we slightly modify the assignment such that $\forall q \in Q$, we find its second closest object r in $(Q \cup R)$ by removing the same object with q and create the edge $q \rightarrow r$. We set the weight of edge $q \rightarrow r$ to distance $|q, r|$. For example, in Fig. 7, we create an edge from each query to its closest object and label the weight of the edge using their distance.

In order to generate a tighter p_q, a good pivot r from $(Q \cup R)$ should capture the following two properties.

- r has as many incoming edges as possible, i.e., the number of queries taking r as their second closest object should be as large as possible. Let $\mathcal{Q}(r)$ be the queries in Q that take r as its second closet pivots, i.e., $\forall q \in \mathcal{Q}(r)$, $\forall r' \in R$, $|r', q| \geq |r, q|$. For simplicity, we write $\mathcal{N}(r) = |\mathcal{Q}(r)|$, and let $\mathcal{N}_{min} = \min_{r \in (Q \cup R)} \mathcal{N}(r)$ and $\mathcal{N}_{max} = \max_{r \in (Q \cup R)} \mathcal{N}(r)$.
- the overall distances from objects in $\mathcal{Q}(r)$ to r are as small as possible. Let $AVG(r)$ be the average distance from $q \in \mathcal{Q}(r)$ to r, i.e., $AVG(r) = \frac{\sum_{q \in \mathcal{Q}(r)} |q,r|}{\mathcal{N}(r)}$. Let $AVG_{min} = \min_{r \in (Q \cup R)} AVG(r)$, and $AVG_{max} = \max_{r \in (Q \cup R)} AVG(r)$.

We define the ranking score $score(r)$ of r by properly leveraging and normalizing $\mathcal{N}(r)$ and $AVG(r)$ below:

$$score(r) = \frac{\mathcal{N}(r) - \mathcal{N}_{min}}{\mathcal{N}_{max} - \mathcal{N}_{min}} + \left(1 - \frac{AVG(r) - AVG_{min}}{AVG_{max} - AVG_{min}}\right) \tag{12}$$

Note that if $\mathcal{N}(r)$ is 0, we simply set $score(r)$ to 0. By computing $score(r) \forall r \in (Q \cup R)$, we then extract L objects with the maximum ranking scores as pivots.

Example 5 (Approximate Pivot Selection) Suppose we have built the bipartite graph shown in Fig. 7 regarding a query set Q and a data set R. First, $\forall r \in R \cup Q$, we set $\mathcal{N}(r)$ by counting the number of queries taking r as their second closest object. In this way, we have $\mathcal{N}_{min} = 0$ and $\mathcal{N}_{max} = 4$. Second, $\forall r \in R \cup Q$, we compute the averaged distance from r to the queries that take r its second closest object. Take r_1 for instance. $AVG(r_1) = \frac{|q_1,r_1|+|q_2,r_1|}{|\mathcal{N}(r_2)|} = \frac{1+3}{2} = 2$. Similarly, we can compute $AVG(r)$ for the remaining objects where we have $AVG_{min} = 0$ and $AVG_{max} = 3$. Third, $\forall r \in R \cup Q$, we compute $score(r)$ based on Eq. 12. For instance, $score(r_1) = \frac{2-0}{4-0} + (1 - \frac{2-0}{3-0}) = 5/6$. We list $\mathcal{N}(r), AVG(r), score(r)$ for each $r \in Q \cup R$ in Fig. 7. Finally, L objects with the largest scores are taken as pivots. □

Details of the one-pass bipartite-based approach using SQL in PostgreSQL are given below:

```
1: SELECT ID AS SID, A
        INTO S
        FROM Q
        ORDER BY random()
        LIMIT m;
2: SELECT ID AS TID, A
        INTO T
        FROM R
        ORDER BY random()
        LIMIT n;
3: ALTER TABLE S ADD pidDist iPair;
4: UPDATE S SET S.pidDist =(
     SELECT (T.PID, DIST(S[A], T[A]))::iPair as pair
     FROM T
     ORDER BY pair.value
     LIMIT 2,1);
5: SELECT pidDist.key as PID, count(*) AS N, 1.0 *
sum(pidDist.value)/count(*) AVG
   INTO SC
   FROM S
   GROUP BY PID;
6: SELECT PID, 1.0*(N - N_min)/(N_max-N_min) + 1 - 1.0 *
(AVG - AVG_min)/(AVG_max - AVG_min) AS score
   INTO RANKING
   FROM SC
   WHERE N > 0
   ORDER BY score DESC
   LIMIT L;
7: SELECT PID, A
   INTO PTable
   FROM T, RANKING
   WHERE ranking.PID = T.PID;
```

To achieve a good performance, we generate two samples, $\mathbb{S}$ for Q and $\mathbb{T}$ for R,[5] respectively (Line 1–2). We then take $\mathbb{S}$ as the query load, $\mathbb{T}$ as the superset of selected pivots, and extract L pivots from $\mathbb{T}$ (Line 3–6). To do this, for each query $q \in \mathbb{S}$, we determine its second closest object from among the objects in $\mathbb{T}$ and maintain the information in an additional column pidDist (Line 3–4). We compute $\mathcal{N}(r)$ and $AVG(r)$ for each $r \in \mathbb{T}$ and maintain the intermediate results in a temporary table SC (Line 5). Finally, we compute the ranking score of each object in table $\mathbb{T}$ and select the L objects with largest scores as the pivots $\mathbb{P}$ (Line 5–7). For ease of illustration, we list the main steps as follows:

1. We scale down the computation by sampling $(Q \cup R)$. Specifically, we generate a sample $\mathbb{T}$ from $R \cup Q$ and another sample $\mathbb{S}$ from Q. We then take $\mathbb{S}$ as the query load, $\mathbb{T}$ as the superset of the pivots;
2. We compute the ranking score for each object $r \in \mathbb{T}$ based on Eq. 12 and the L objects with greatest ranking scores will be selected as the pivots $\mathcal{P}$.

The time complexity of using this heuristic approach to selecting the pivots is $O(|\mathbb{S}| * |\mathbb{T}| * \Delta + |\mathcal{P}| * \log(|\mathbb{S}|))$ where Δ is the averaged distance computation cost regarding R. In this paper, we implicitly assume that the size of the data set R is fairly large, and hence extracting effective pivots from R is practical. This assumption is reasonable because if $|R|$ is small, it is anyway unnecessary to build an index for similarity queries.

Discussion over the query distributions In reality, all existing approaches to selecting pivots make an implicit [18,25,34] or explicit [23,28] assumption: queries follow the same distribution as the data set. For example, by making the above assumption, iDistance and its variants partition the data sets (queries) into clusters and extract the centers as the pivots. On the contrary, if queries do not follow the same distribution as the data set and in extreme cases that queries are all far away to the data set, then applying k-means algorithm to the original data set is not a good pivot selection approach since pivots are expected to be close to queries. While in our paper, we generalize the query distribution as any discrete distribution, our approach is adaptive to select pivots as long as the query distribution is given aforehand. Initially, when the query load is not available, we assume that queries follow the same distribution as the data set. We then select $|P|$ using the above-presented one-pass bipartite-based approach. It is worth mentioning that the query performance can be degraded when the query distribution changes. To address this issue, we propose an incremental pivot update mechanism described below:

1. Likewise, we generate a sample $\mathbb{T}$ from $R \cup Q$ and another sample $\mathbb{S}$ from Q.
2. We then refine $\mathbb{P}$ using objects in $\mathbb{T}$ incrementally. Under the new query load, we compute $score(p)\ \forall p \in \mathbb{P}$, and sort pivots in $\mathbb{P}$ based on the ascending order of their scores. $\forall r \in \mathbb{T}$, we compute $score(r)$ as well. We sort objects in $\mathbb{T}$ but based on the descending order of their scores. Next, we incrementally replace $p_i \in \mathbb{P}$ using $r_i \in \mathbb{T}$ if $score(r_i) > score(p_i)$, where $i \in [1, |\mathbb{P}|]$. Considering that it is computational expensive to repartition the data set if too many pivots are changed, an alternative is to update pivots by setting a percentage.
3. Finally, we re-assign objects in the obsolete partitions, i.e., partitions whose corresponding pivots are replaced, and update their signatures accordingly.

Besides, when the number of records in R changes by a certain percentage (e.g., the number of records in R increases by 10%), it is also necessary to invoke an incremental pivot update mechanism. In this case, we can introduce more pivots in order to reduce the number of candidates. Specifically, we run the proposed heuristic approach to selecting an extra set of pivots from the newly added records, repartition the records, and update their signatures accordingly.

To verify the effectiveness of our incremental pivot update mechanism, we conduct the evaluation in Sect. 5.2.4.

5 Experiments

In this section, we first describe the configurations of our experiments, including evaluation data sets, similarity functions, methods, and performance metrics. We then study the key parameters that potentially affect the performance of MSQL and compare MSQL with the state-of-the-art approaches.

5.1 Experimental setup

Data sets and similarity functions We evaluate the query performance over 6 real data sets belonging to the following three application domains:

– *Spatial databases* We use 2 spatial data sets. OpenStreetMap[6] is a world-wide geographic data set that collects a multitude of points of interest, buildings, natural features and land-use information. Forest[7] predicts forest cover types from cartographic variables. We use L1 and L2 as the similarity functions and set L1 as the default function.

[5] An alternative of $\mathbb{T}$ extraction is first to union R and Q, and then do the random extraction.

[6] https://www.openstreetmap.org/.

[7] http://archive.ics.uci.edu/ml/datasets/Covertype.

– *Text retrieval* We use 3 string data sets, including Author, Actor, and Movie. Author data set consists of author names extracted from DBLP.[8] Actor and Movie data sets consist of actor names and movie titles extracted from IMDB.[9] We use edit distance, Jaccard, Cosine as the similarity functions.
– *DNA sequence* Uniprot data set consists of protein sequences in flat text format.[10] We use edit distance as the similarity function.

Table 5 shows the detailed statistics of the data sets. It can be seen that for string data sets the cardinality ranges from 373,969 to 3,798,125. Note that we have removed duplicates from the data set in order to eliminate the factors that may potentially affect the analysis of our methods. The averaged string length of the three string data sets is 15.04, 14.46, and 18.74, respectively, while the variance of them is only 3.34, 3.86, and 9.54, respectively. This observation presents the distribution of string length of the three data sets is fairly dense. On the contrary, the DNA sequences in Uniprot has large averaged sequence length as well as large variance, i.e., the distribution of sequence length of Uniprot is sparse.

Comparative approaches We compare MSQL with the following state-of-the-art approaches:

– *Reference-based approaches* In this paper, our objective is to propose a solution that is generic and portable across both open source and commercial RDBMS. However, as discussed in Sect. 2.3, fully integrating reference-based approaches as a generic and portable solution into RDBMS is infeasible. Instead, we abstract the core techniques from them and propose the **baseline** approach and the pivot selection methods are listed in Table 2.
– *Native solutions* There exist a large number of native solutions for processing similarity queries. For simplicity, we select some representatives that are widely used or recently published. There are PBI, B^{ed}-tree, and Flamingo for text spaces, iDistance for Euclidean spaces, SPB-tree, M-tree for metric spaces. We do not plot the results of any comparative approach if it fails to return the results in an acceptable time or I/O times.
– *SQL-based solutions* GiST [16] is a built-in index in PostgreSQL, and is well recognized as a generalized index structure supporting an extensible set of queries and data types. By default, GiST is used to implement R-tree, supporting spatial queries over Euclidean spaces in PostgreSQL. For ease of illustration, we use notation **GiST-RTree** instead of GiST. z-kNN [31],

Table 5 Statistics of used data sets

Data set	Similarity Function	Cardinality	Size (MB)	Min. Len	Max. Len	Avg. Len	Var.	# of Dims	Threshold θ
Actor	**Edit distance** Jaccard, Cosine	1,213,290	28	1	72	15.04	3.34	/	0,1,**2**,3,4
Author	Edit distance	3,798,125	89	4	47	14.46	3.86	/	0,1,**2**,3,4
Movie	Edit distance	373,969	9.9	1	239	18.74	9.54	/	0,1,2,3,**4**,5,6
OpenStreetMap	**L1**,L2	100,000,000	271	/	/	/	/	2	0,0.01,**0.02**,0.04,0.08,0.1,0.2
Forest	**L1**,L2	581,012	263	/	/	/	/	10	100,**200**,300,400,500,600,1000
Uniprot	Edit distance	508,038	170	2	1992	341	251.05	/	0,2,**4**,8,16

[8] http://www.informatik.uni-trier.de.
[9] http://www.imdb.com.
[10] http://www.uniprot.org.

Table 6 Native and SQL-based solutions to be compared

	Euclidean spaces	Text spaces	Metric spaces	SQL-based
GiST-RTree	√			√
z-kNN[31]	√			√
iDistance[18]	√			
Text-SQL[15]		√		√
PBI[23]		√		
B^{ed}-tree[36]		√		
Flamingo[4]		√		
SPB-tree[8]	√	√	√	
M-Tree[9]	√	√	√	
MSQL	√	√	√	√

originally designed for processing k nearest neighbor queries over *Euclidean spaces* using SQL, is extended for processing similarity queries over *Euclidean spaces*. Text-SQL [15] is the only known method to process string similarity search under edit distance metric using SQL.

Table 6 surveys native solutions and SQL-based solutions. Note that MSQL is the only SQL-based solution to process similarity queries in metric spaces. All source code of comparative approaches has been generously provided by the authors or downloaded from the home pages of the authors. We evaluate their performance in terms of average elapsed time, average candidate set size and average I/O times by repeating each experiment for five times. We randomly select 100 query objects from each data set and compare the average elapsed time. We list the thresholds that are commonly set in existing work across various data sets in Table 5; we use the threshold highlighted in **bold** as default in the experiments. All experiments are conducted on a PC with Intel E5620 2.4GHz of CPU, 8GB of memory, and CentOS 5.5 operating system. All experiments of GiST-RTree, z-kNN, Text-SQL, MSQL are executed in PostgreSQL 9.4.5. We use default parameters in PostgreSQL, of which each page size is set to 8192B and the number of shared buffers in the buffer pool is set to 1000 (Fig. 8).

5.2 Evaluation of MSQL

We evaluate the effectiveness of the new pruning rule and our pivot selection strategy, compare MSQL with reference-based approaches in terms of the performance of query processing and data maintenance, and analyze the index construction time. We leave the number of pivots as a tuning parameter, which varies in {500, 1000, 2000, 4000, 8000} for Actor, Author, Movie and OpenStreetMap data sets, {100, 200, 300,400, 500} for Forest data set, and {10, 25, 50, 100, 200} for Uniprot data set.

5.2.1 Effect of pruning rules and pivot selection

We study the performance of MSQL under Theorem 1 (a.b.a **T1**) and Theorem 3 (a.b.a. **T3**). Besides the pivot selection strategies listed in Table 2, we also use the **Heuristic** strategy that is proposed in Sect. 4.2.2 and **k-medoids** clustering method (resp. k-mean) to select the pivots. For **Heuristic, k-mean, k-medoids, MaxProb and MaxVar**, we do sampling by randomly selecting 1000 objects as $\mathbb{Q}$ and 2000 objects as $\mathbb{T}$ in Uniprot and 10,000 objects as $\mathbb{Q}$ and 20,000 objects as $\mathbb{T}$ in each of remaining data sets.

We first plot the number of candidates in Fig. 9a–c. When the number of pivots increases, the candidate set size decreases monotonously while there is a clear trend of diminishing gains. Because Theorem 3 produces tighter search ranges than Theorem 1, as we can see, **T3 + Random** always generates less number of candidates than **T1 + Random**. Besides, because it is able to select better quality of pivots, the number of candidates using **T3 + Heuristic** is less than that using **T3 + Random** by a factor ranging from 31 to 46% when $|\mathbb{P}|$ is small. However, there is a clear trend of diminishing gains when $|\mathbb{P}|$ increases. The reason, as explained in Sect. 4, is that while there are sufficiently many pivots, it is likely that for each query, we can find a close pivot in both **T3 + Heuristic** and **T3 + Random**, leading to a similar pruning power. Interestingly, in most cases, **T3 + MaxProb** and **T3 + MaxVar** even generate larger number of candidates than **T3 + Random** and **T3 + Heuristic**. As discussed in Sect. 2.2, the cost models of selecting the pivots in MaxProb and MaxVar are under T1. Often, as pointed out in [23,28], objects are typically assigned to a distant pivot so that the query processing is optimized. While in MSQL, according to Theorem 3, we expect to find a pivot with the minimum distance to q, for this reason, **T3 + MaxProb** and **T3 + MaxVar** lead to larger search ranges than **T3 + Random** and **T3 + Heuristic**. k-means and k-medoids generate slightly smaller number of candidates than **T3 + Heuristic**, but the benefit drops smoothly when $|\mathbb{P}|$ increases. k-medoids only work in Uniprot (the sample is small), and it fails to select pivots in other data sets with an acceptable time.

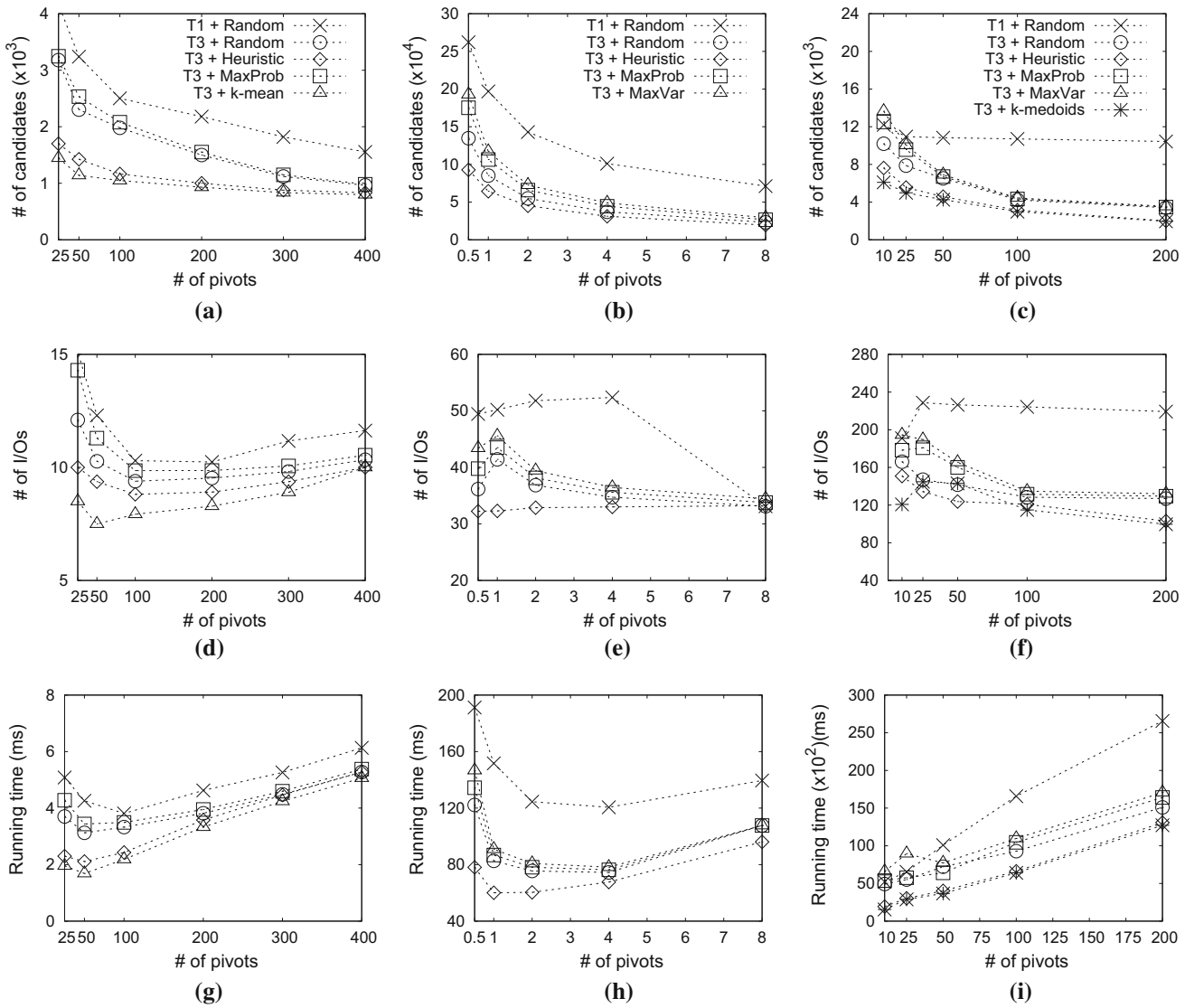

Fig. 8 Effect of pivot selection strategies **a** Forest and candidate size **b** Actor and candidate size **c** Uniprot and candidate size **d** Forest and I/Os **e** Actor and I/Os **f** Uniprot and I/Os **g** forest and query time **h** Actor and query time **i** Uniprot and query time

We then analyze the I/O cost and plot the results in Fig. 9d–f. As we can see, **T3 + k-means**, **T3 + k-medoids** and **T3 + Heuristic** generally consume the minimum I/O cost, followed by **T3 + Random**, **T3 + MaxProb**, **T3 + MaxVar** and **T1 + Random**. In MSQL, the I/O cost depends on the number of pages that the candidates reside in. Often, a larger number of candidates lead to a more expensive I/O cost (see Fig. 9d, e). But in some cases, if the candidates are scattered across quite a more different set of pages, these candidates will consume more expensive I/O cost (see Fig. 9f).

Finally, we study the running time and plot the results in Fig. 9g–i. We make two observations. First, the trend of the running time generally follows that of the number of candidates as well as the number of I/Os. That is, **T3 + k-means**, **T3 + k-medoids** and **T3 + Heuristic** perform the best, followed by **T3 + Random**, **T3 + MaxProb**, **T3 + MaxVar** and **T1 + Random**. Second, when the number of pivots varies, the running time first decreases, but then increases after the number of pivots exceeds a certain threshold, which is 1000 for Actor, 50 for Forest, and 10 for Uniprot, respectively. This is due to the fact that the running time in MSQL mainly consists of three parts: (1) distance computation cost of pivots to the query, (2) I/O cost of loading candidates from the local disk to main memory, and (3) verification cost of candidates to similar objects. In our case, Uniprot takes the most expensive distance computation cost, followed by Actor and Forest. By increasing the number of pivots, the cost of distance computation increases linearly. On the contrary, using an increasing number of pivots leads to a less number of candidates. Without loss of generality, by introducing a new pivot p, let DC

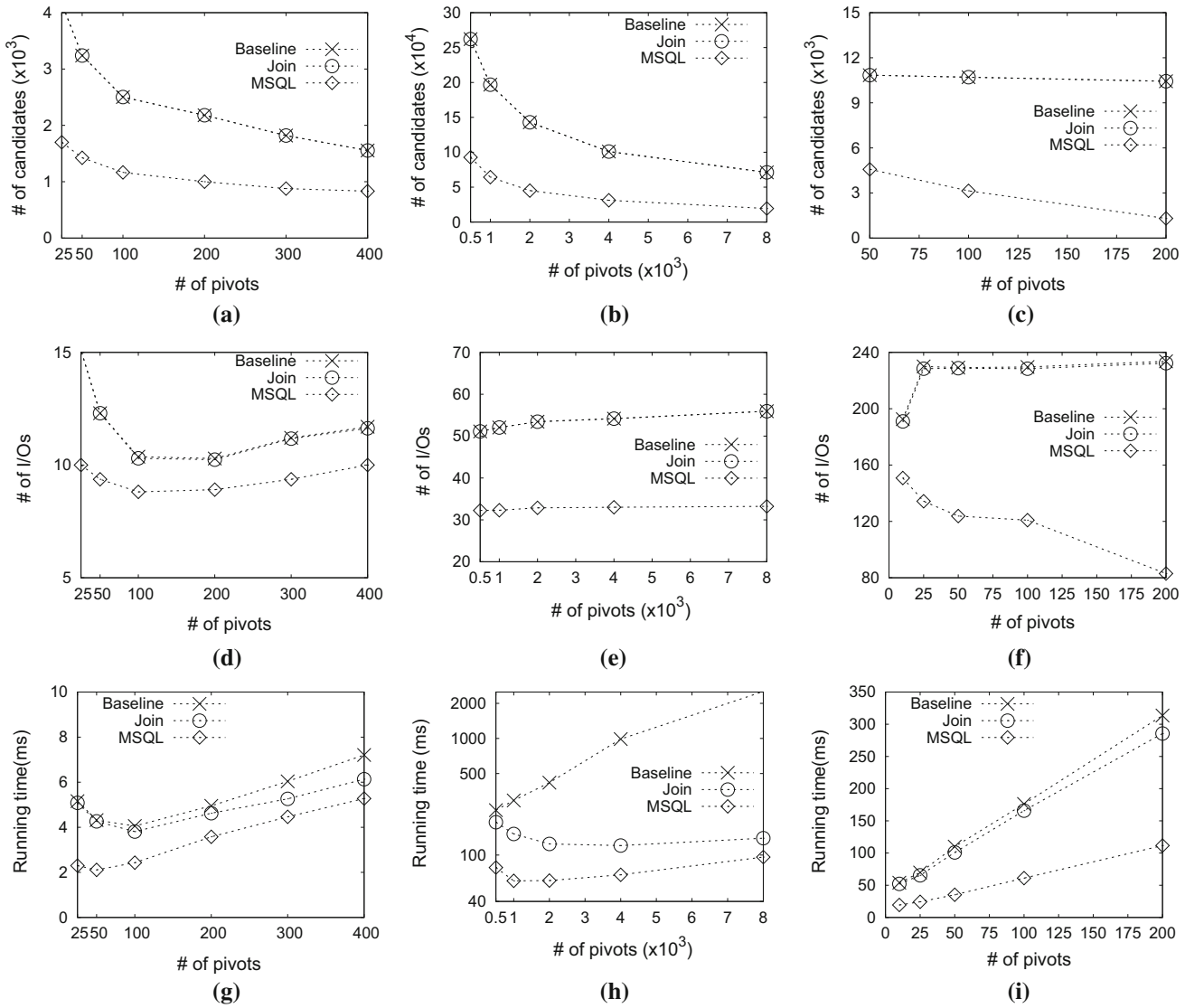

Fig. 9 Comparing MSQL with reference-based approaches using SQL **a** Forest and candidate size **b** Actor and candidate size **c** Uniprot and candidate size **d** Forest and I/Os **e** Actor and I/Os **f** Uniprot and I/Os **g** Forest and query time **h** Actor and query time **i** Uniprot and query time

be the extra distance computation cost, δ be the objects that is filtered out merely based on p, and VC be the saving cost that verifies whether $r \in \delta$ is similar to q or not. As shown in Fig. 9a–c, there is a clear trend of diminishing gain of candidate reduction when $|\mathbb{P}|$ increases, while DC remains almost invariant. For this reason, the running time drops when $VC > DC$, and increases when $DC > VC$. Because VC in Uniprot is fairly expensive, the optimal number of pivots is the smallest among six real data sets.

Similar experiments are conducted over the remaining data sets, and we set $|\mathbb{P}| = 1000$ for Author, Movie, OpenStreetMap. Besides, due to applicability and performance issue, we set MSQL under **T3 + Heuristic** in the remaining experiments.

5.2.2 Comparison with reference-based approaches

We compare MSQL with the baseline approach (a.b.a Baseline), shown in Sect. 3.4.1, and the Join-based approach (a.b.a. Join), shown in Sect. 3.4.2. Recall that Baseline and Join adopt the pruning rules in T1, while MSQL adopts the pruning rule in Theorem 3.

The results are plotted in Fig. 9. First, Baseline and Join generate the same number of candidates and almost the same number of I/Os, while Join performs faster than Baseline, and the superiority of Join becomes more obvious when the number of pivots increases. The reason, as discussed before, is twofold. On the one hand, selecting the best query plan in the query optimizer for Baseline is costly when the number of predicates is large. On the other

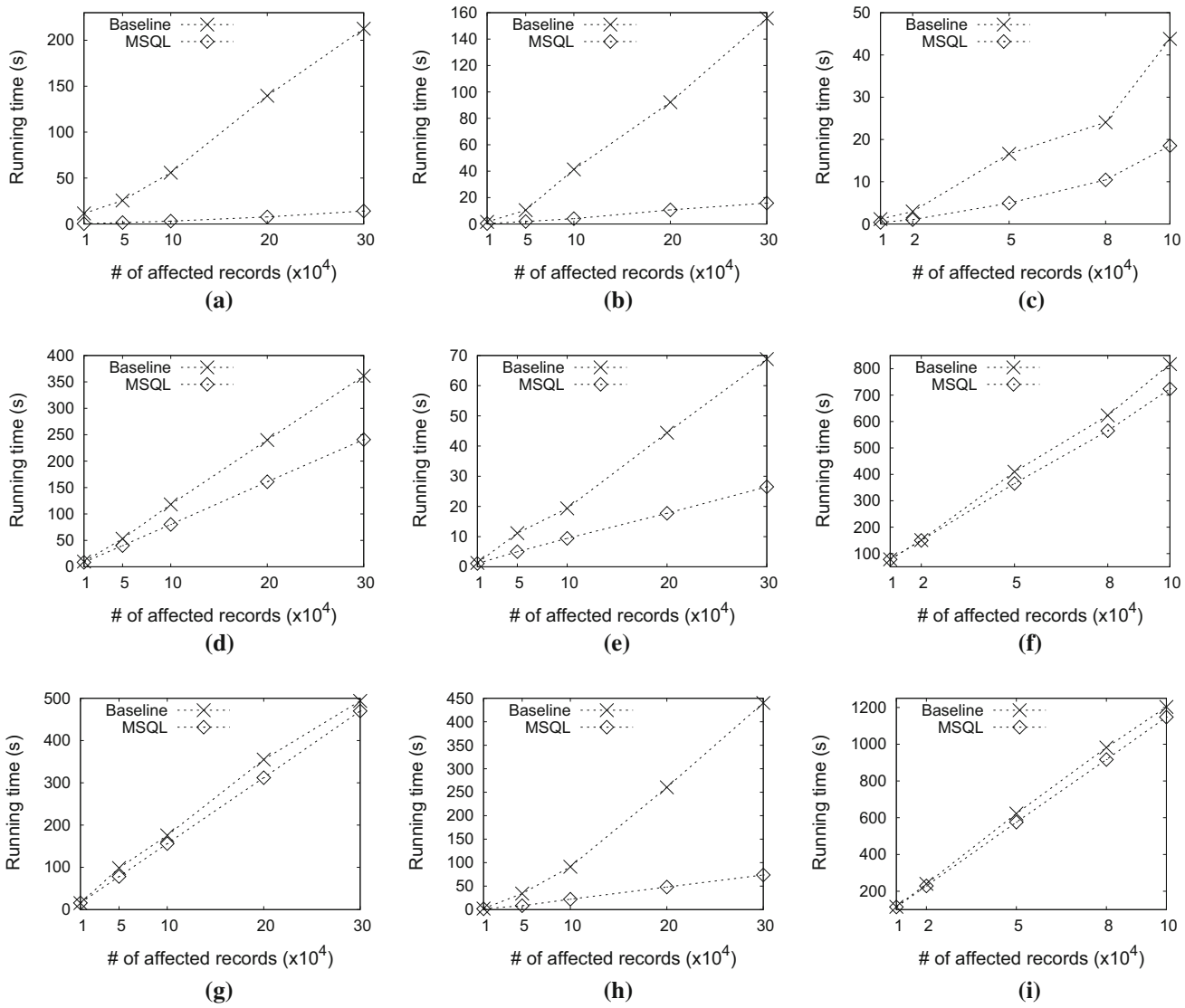

Fig. 10 Effect of data manipulations **a** Actor and data deletion **b** Forest and data deletion **c** Uniprot and data deletion **d** Actor and data insertion **e** Forest and data insertion **f** Uniprot and data insertion **g** Actor and data update **h** Forest and data update **i** Uniprot and data update

hand, when the number of predicates is large, the query optimizer in Baseline resorts to table scan instead of index seeks, nullifying the effect of index seek (see Fig. 9h). Second, the number of candidates in MSQL is reduced by up to a factor of 87.5% in comparison with that in Baseline and MSQL always generates less number of candidates than Baseline (Join) since Theorem 3 produces tighter search ranges than Theorem 1 as well as effective pivots are used. Third, the I/O cost is reduced significantly by MSQL in comparison with Baseline due to the effectiveness of the pruning rule, i.e., Theorem 3. Fourth, MSQL runs faster than Join and Baseline due to the less number of candidates and I/Os.

5.2.3 Data maintenance

We compare MSQL with Baseline in terms of data maintenance cost. Note that it is necessary to maintain $S_i.min$ and $S_i.max$ in **PTable**, while MSQL does not.

We first study the performance of handling data insertion, update, and deletion. Figure 10 plots the results over Actor, Forest, and Uniprot by inserting, updating, and deleting the number of records from 10,000 to 300,000 for Actor and Forest and from 10,000 to 100,000 for Uniprot. As we can see, for data deletion, MSQL performs up to two orders of magnitude faster than Baseline. The reason is that Baseline needs to examine $S_i.min$ and $S_i.max$ by computing

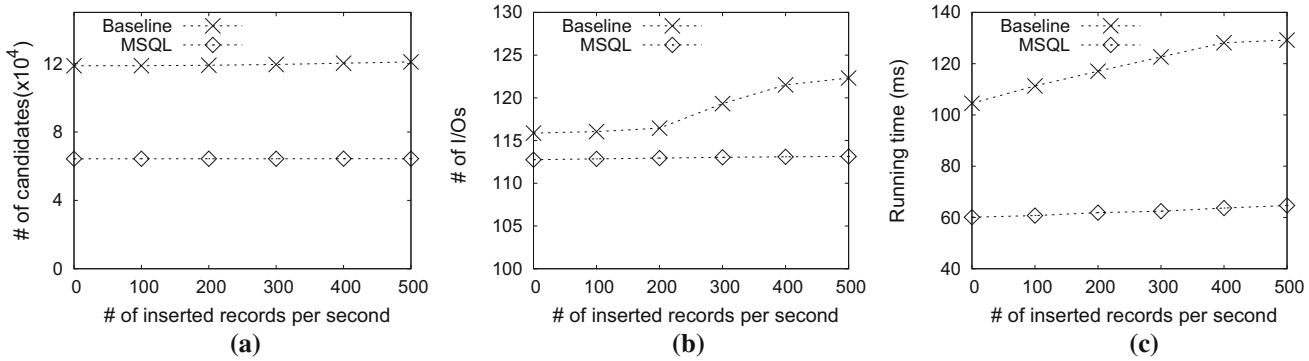

Fig. 11 Concurrent query processing **a** # of candidates **b** # of IOs **c** running time

$|r, p_i|$ (suppose r belongs to P_i^R), and replace it with the new $S_i.min$ and $S_i.max$ if necessary. While MSQL does not necessary conduct this examination and replacement, thus such computation cost can be completely saved. For data insertion and data update, as MSQL also needs to compute the signatures of new objects, the benefit of MSQL in these two cases is not as good as that in data deletion. But still, MSQL performs 1.1–8× faster than Baseline. From Fig. 10d–i, we observe that the smaller the signature computation cost is, the more the benefit of MSQL will gain. The reason is that, in our experiments, the sequence space takes the most expensive signature computation cost, followed by the text space and Euclidean space. When the signature computation cost dominates the index update cost, the performance of MSQL is close to that of Baseline (see Fig. 10i).

MSQL supports concurrent query processing that other native solutions cannot. For this reason, we study the query performance by issuing continuous similarity queries as we simultaneously insert the number of records (from 0 to 500 per second) into the original relation. We plot the result in Fig. 11 over Actor data set. As we can see, compared with Baseline, although MSQL produces less than half the number of candidates as well as <20% I/O cost, MSQL performs 3X faster than the baseline approach. The reason, as discussed before, is that MSQL eliminates extra maintenance cost of $S_i.min$ and $S_i.max$ and concurrent locking issue over PTable. Besides, when the frequency of data insertion increases, compared with Baseline, there is a clear trend of increasing grain using MSQL.

5.2.4 Incremental pivot update

In this section, we evaluate the effectiveness of our proposed incremental pivot update mechanism over Author and Forest data sets.

We first study the effect when the query load varies. In this setting, we generate two query sets, in which the first is randomly extracted from the first 50% of each data set, and the other is randomly extracted from whose data set. The results are given in Table 7. As we can see, for Author, among 1000 pivots, 814 pivots are replaced in order to achieve the minimum candidate set size regarding the new query load, and the number of candidates under the refined pivots is reduced by a factor of 13%. For Forest, among 50 pivots, 41 pivots are replaced and the number of candidates under the refined pivots is reduced by a factor of 8% (Table 8).

Table 7 Incremental pivot update for dynamic query load

Data set	# of pivots	# of replaced pivots	# of reduced candidates (%)
Author	1000	814	13
Forest	50	41	8

Table 8 Incremental pivot update for dynamic data sets

		40%	60%	80%	100%
Author	Time	0	0.14	0.21	0.35
	I/O	0	0.08	0.24	0.33
	Candidates	0	0.21	0.28	0.36
Forest	Time	0	0.10	0.22	0.30
	I/O	0	0.06	0.12	0.16
	Candidates	0	0.19	0.28	0.35

We then investigate the effect when the data set changes. In this setting, we first randomly extract 40% of objects in each data set, and select 40% of total pivots. We then run our heuristic approach to selecting extra 20% pivots when 20% of objects from remaining data set are added. We compare the query performance under the dynamic pivots and static pivots (the pivot set is fixed when the data set changes) and show the benefit that is saved by our proposed incremental pivot update mechanism. As we can see, the query time, the number of I/Os, and the number of candidates can be reduced by a factor up to 35, 33, 36% respectively when 40, 60, 80, and 100% of objects in the data sets are used.

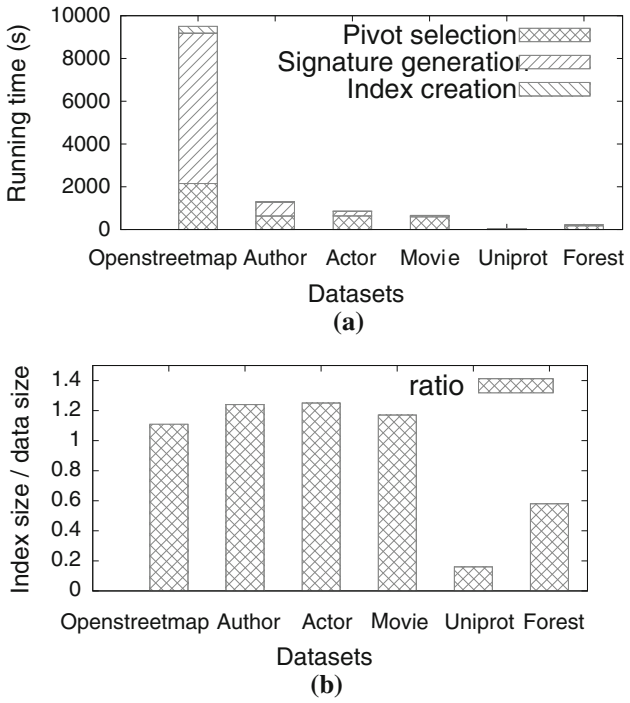

Fig. 12 Index building **a** index construction time **b** index size and data size

5.2.5 Index building

Figure 12a shows the index construction time over six real data sets. Generally, following our index generation scheme, the index construction time consists of three parts: (1) pivot selection, (2) signature generation, and (3) B^+-tree construction. As discussed in Sect. 4, the time complexity of pivot selection is $O(|\mathbb{S}| * |\mathbb{T}| * \Delta + |\mathcal{P}| * \log(|\mathbb{S}|))$. For the data sets except Uniprot, as we use the same size of $|\mathbb{S}|$ and $|\mathbb{T}|$, the running time of pivot selection phase relies on Δ, the averaged distance computation cost (Note that $|\mathcal{P}| * \log(|\mathbb{S}|$ is typically much less than $|\mathbb{S}| * |\mathbb{T}| * \Delta$). That is the reason why in Fig. 12a, regarding the pivot selection phase, OpenStreetMap (OSM for short in the figure) takes the minimum running time, followed by Forest, Author, Actor, Movie, and Uniprot. For signature generation phase, its time complexity is $O(|\mathbb{P}| * |R| * \Delta)$ and hence the running time of this phase relies on the pivot size $|\mathbb{P}|$, data set size $|R|$ and the averaged distance computation cost Δ. From Fig. 12a, we can observe that OpenStreetMap takes maximum signature generation time, followed by Author, Actor, Forest. The time complexity of B^+-tree construction is $O(|R|log|R|)$, and hence Movie takes the minimum running time of this

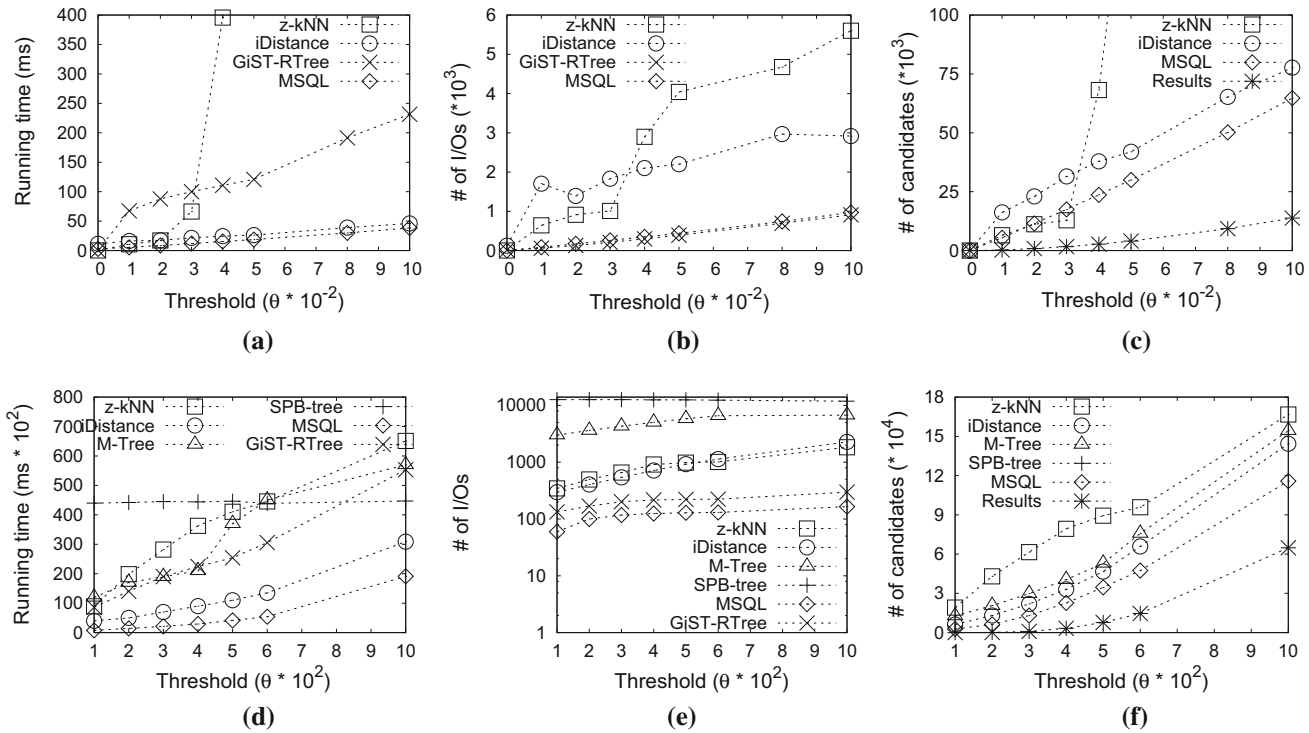

Fig. 13 Similarity search in Euclidean spaces **a** OpenStreetMap and query time **b** OpenStreetMap and I/Os **c** OpenStreetMap and candidate size **d** Forest and query time **e** Forest and I/Os **f** Forest and candidate size

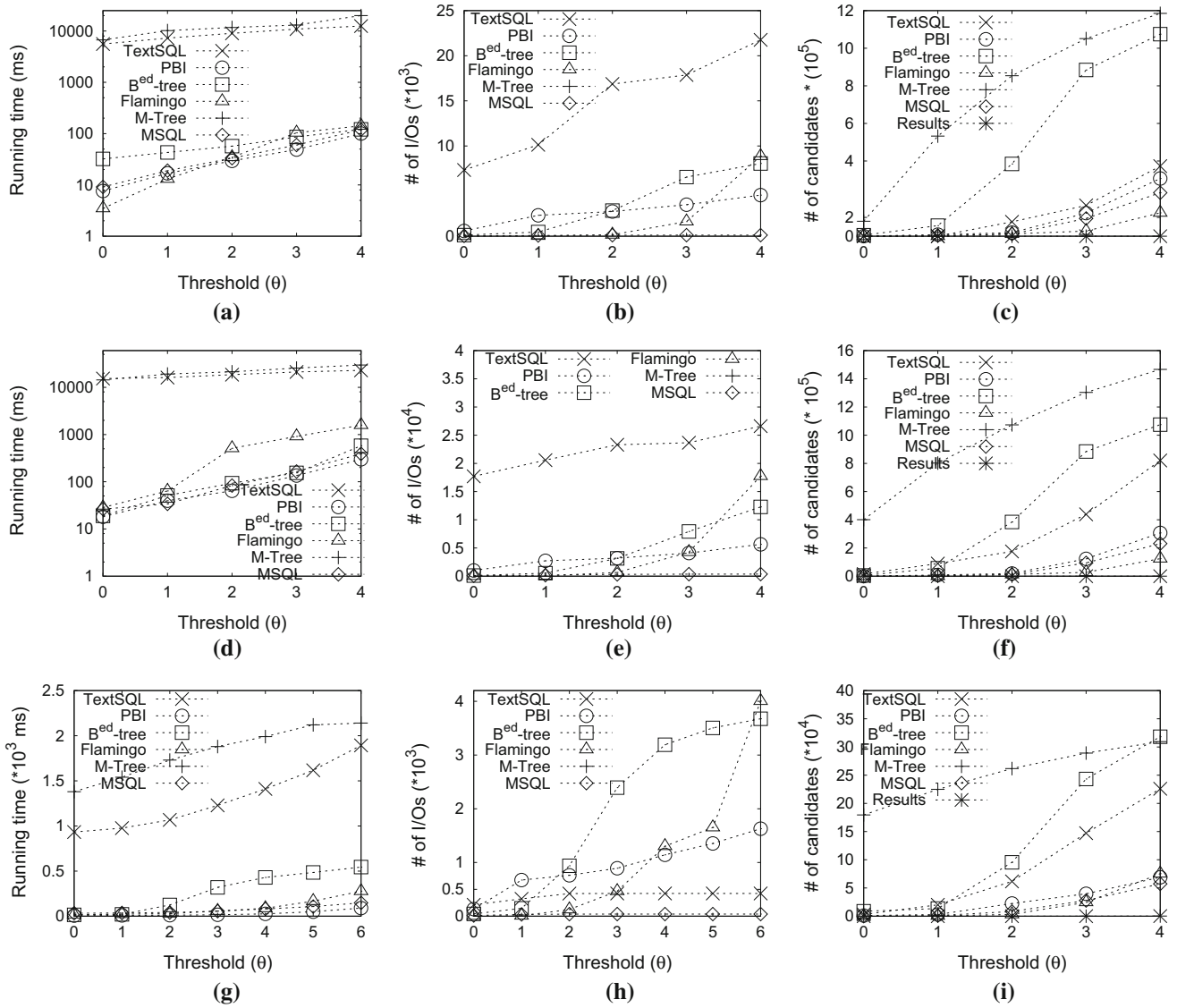

Fig. 14 Similarity search in text spaces **a** Actor and query time **b** Actor and I/Os **c** Actor and candidate size **d** Author and query time **e** Author and I/Os **f** Author and candidate size **g** Movie and query time **h** Movie and I/Os **i** Movie and candidate size

phase, followed by Uniprot, Forest, Actor, Author, and OpenStreetMap. By taking these three parts into account, Uniprot consumes the minimum index construction time, followed by Forest, Movie, Actor, OpenStreetMap, Author. Although it takes more than 1 h to build the index in OpenStreetMap (100,000,000 objects), we argue that existing RDBMS like PostgreSQL 9.6 and its later versions[11] start to support parallel execution of SQL statements, while considering the process of building index in MSQL, it is able to performs this parallelism by dividing the data set into multiple subsets, and parallelizing the computation over each subset separately.

Figure 12b shows the ratio (index size / data size). As we can see, the ratio ranges from 1.1 to 1.25 for OpenStreetMap, Author, Actor, and Movie since the averaged length of objects is comparable to the size of signatures. While the averaged length of objects is much larger than the size of signatures, we observe that the ratio degrades from 0.17 to 0.6 for Uniprot and Forest.

5.3 Comparison with existing approaches

We compare MSQL with SQL-based and native solutions listed in Table 6 for Euclidean spaces, text spaces and sequence spaces separately.

[11] https://www.postgresql.org/docs/9.6/static/release-9-6.html.

5.3.1 Euclidean spaces

Figure 13 shows the query performance of comparing MSQL with existing solutions using L2. Note that *GiST-RTree insides PostgreSQL integrates the candidate identification phase and verification phase together. For this reason, we do not plot the candidate set size of GiST-RTree. Instead, we plot the size of final results.*

• *Comparison with SQL-based approaches* In general, MSQL performs the best, followed by GiST-RTree and z-kNN. For z-kNN, it loses partial locality information by mapping multi-dimensional point to one-dimensional value and hence it generates much larger number of candidates than both GiST-RTree and MSQL. GiST-RTree consumes less number of I/Os, and runs faster than MSQL when $\theta = 0$. Nevertheless, this is later reversed and the gain of using MSQL is enhanced as the threshold varies. MSQL runs up to 6X faster than GiST-RTree when $\theta = 0.1$. GiST-RTree consumes slightly less number of I/Os than MSQL. As frequently pointed out in existing literatures, GiST-RTree suffers from so-called curse of dimensionality [12,13], and is only suitable for low-dimensional spaces, especially for the data with 2–4 dimensions. For this reason, over the high-dimensional space (Forest), we find that MSQL runs 5.65–11.3× faster than GiST-RTree and requests less number of I/Os by 1.7–3.89×.

• *Comparison with native solutions* Interestingly, MSQL performs the best in terms of CPU time, I/O cost, and candidate set size. The reason is that, by selecting sufficiently many pivots to split the data space into small cells, it is common for MSQL to find a close pivot for each query, which helps generate a tight upper bound of each search range. Due to a larger number of candidates and I/Os, iDistance performs slightly slower than MSQL. SPB-tree transforms the similarity queries into range queries over L-dimensional spaces. Apparently, L, the number of used pivots, cannot be too large; otherwise, the similarity query problem is transformed into range queries in high-dimensional spaces, which is widely recognized as an intractable problem. Nevertheless, as discussed in our experiments, using hundreds or even thousands of pivots can achieve the best query performance. Thus, MSQL runs 8–58× faster than SPB-tree, and requests less number of I/Os by two orders of magnitude. For the sake of clarity, we do not show the number of candidates using SPB-tree, which is one to two orders of magnitude larger than that using MSQL.

5.3.2 Text spaces

Figure 14 shows the query performance of comparing MSQL with existing solutions over Actor, Author, and Movie under edit distance.

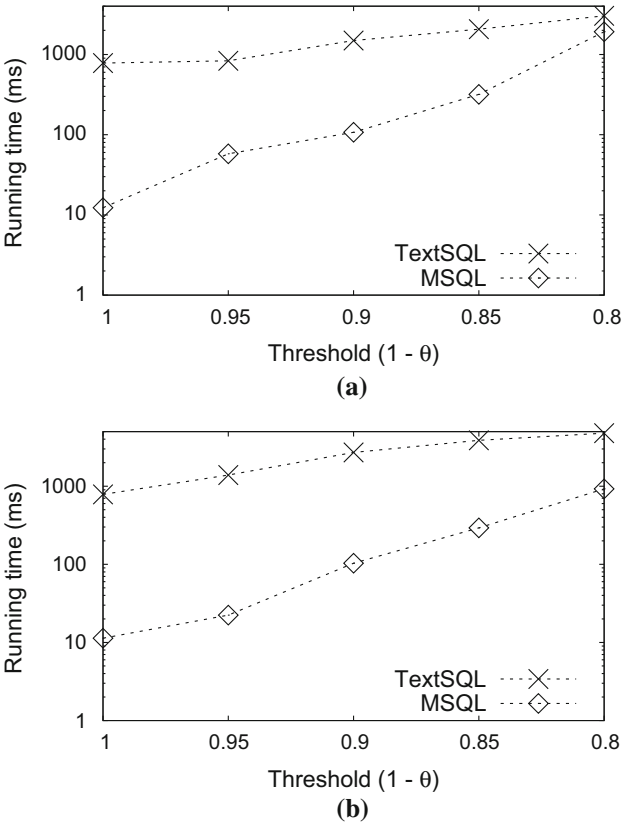

Fig. 15 Actor and Jaccard and Cosine **a** Jaccard and query time **b** Cosine and query time

• *Comparison with SQL-based approaches* MSQL outperforms Text-SQL by two orders of magnitude in terms of both query time and I/O cost, and the performance of MSQL becomes even more substantial when θ varies. The reason is that MSQL determines the candidates by merely examining the signatures of objects located in a set of certain search ranges. In contrast, Text-SQL determines the candidates by issuing a complex quad-table join that involves in two relations with cardinality multiple times larger than that of R, leading to a prohibitive increase of the join cost.

• *Comparison with native solutions* MSQL and native solutions are comparable over all three data sets. In general, MSQL runs faster up to 2.5× faster than B^{ed}-tree, faster up to two order of magnitude than M-Tree, and slower up to 1.1× than PBI. Compared with MSQL, B^{ed}-tree and PBI consume larger number of I/Os by one to two orders of magnitude, and generate larger number of candidates. When the threshold is small, MSQL runs slightly slower than Flamingo which generates less number of candidates and requests less number of candidates I/Os, while this is later reversed when the threshold is relatively large. We observe when the number of candidates increases, the I/O cost is raised significantly in Flamingo. The reason is that Flamingo identifies objects by probing the inverted index and the candidates to be verified are scattered across different inverted lists, making the I/O

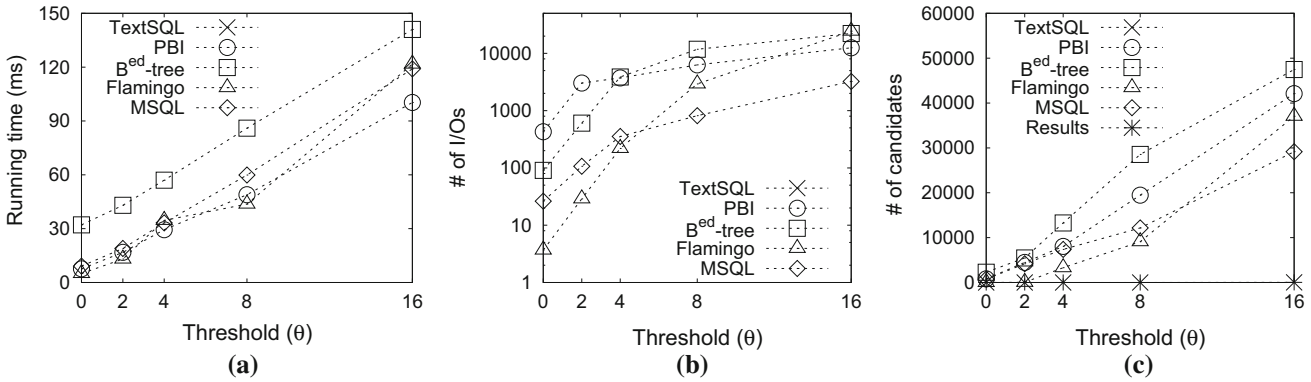

Fig. 16 Similarity search in DNA sequence **a** Uniprot and query time **b** Uniprot and I/Os **c** Uniprot and candidate size

cost prohibitively expensive when the number of candidates is large. Note that in some cases, even Flamingo generates less number of candidates than MSQL and PBI, but still, MSQL and PBI run faster than Flamingo. The reason is that Flamingo takes much more expensive time to figure out the candidates (see [23]).

To measure the similarity between two strings under Jaccard and Cosine, we split each string into a sequence of Q-grams and measure the string similarity based on their Q-grams. We evaluate MSQL with Text-SQL over Actor and plot the experimental results in Fig. 15. Although Text-SQL cannot directly support similarity queries under Jaccard and Cosine, we modify Text-SQL to support both two similarity measures by incorporating the prefix-based pruning techniques [26]. Due to the same reason discussed above, again, MSQL runs faster than Text-SQL by up to two orders of magnitudes. We do not show the I/O cost and candidate set size due to space limitations, but they follow the same general trend as under the edit distance measure.

5.3.3 Sequence spaces

Figure 16 shows the query performance over Uniprot.

• *Comparison with SQL-based approaches* As in sequence spaces, MSQL performs significantly faster than Text-SQL; in some cases, Text-SQL even fails to complete the query within an acceptable time. As explained above, the reason is that the average length of sequence in Uniprot is 341 (Table 5), and R' enlarges the records in Uniprot by more than 300 times, leading to a prohibitive increase of join cost, whereas MSQL still processes similarity queries by examining the objects with signatures located in a set of certain search ranges only.

• *Comparison with native solutions* MSQL and native solutions are comparable in sequence spaces. In particular, when the threshold is very small (in our case $\theta \leq 2$), Flamingo shows its advantages. When the threshold varies, as in the sequence space, the candidates generated in Flamingo increase sharply, resulting in a comparable performance with MSQL and PBI.

To summarize, in most cases, MSQL outperforms all existing SQL-based solutions in Euclidean, text, and sequence spaces significantly. As a systematic solution, MSQL is comparable to existing native solutions in terms of both CPU time and candidate set size. More importantly, based on the benefits brought by RDBMS and its properties, MSQL requires I/Os than existing native solutions by up to two orders of magnitude.

6 Conclusion

In this paper, we propose **MSQL**, an efficient metric space similarity search solution using SQL. MSQL enables users to submit similarity search queries in arbitrary metric spaces using simple SQL statements. It leverages the optimizations and features of existing RDBMS and explores various query optimization techniques to significantly reduce both CPU and I/O cost. Extensive experiments on PostgreSQL demonstrate the efficiency of MSQL, which performs up to two orders of magnitude faster than existing SQL-based techniques. Compared with native solutions, MSQL is comparable in text and sequence spaces, and superior in Euclidean spaces. Furthermore, as an SQL-based solution, MSQL can be deployed easily to existing RDMBS as well.

Acknowledgements We would like to thank the anonymous reviewers for their helpful and insightful comments. This work was in part supported by the National Natural Science Foundation of China (61502504, 61732014), and the Fundamental Research Funds for the Central Universities, the Research Funds of Renmin University of China under Grant No. 15XNLF09.

References

1. Apers, P.M.G., Blanken, H.M., Houtsma, M.A.W. (eds.): Multimedia Databases in Perspective. Springer, Berlin (1997)

2. Aronovich, L., Spiegler, I.: Cm-tree: a dynamic clustered index for similarity search in metric databases. Data Knowl. Eng. **63**(3), 919–946 (2007)
3. Baeza-Yates, R.A., Ribeiro-Neto, B.A.: Modern Information Retrieval. ACM Press/Addison-Wesley, New York (1999)
4. Behm, A., Ji, S., Li, C., Lu, J.: Space-constrained gram-based indexing for efficient approximate string search. In: Proceedings of the 25th International Conference on Data Engineering (ICDE), March 29 – April 2, 2009, Shanghai, China, pp. 604–615 (2009). doi:10.1109/ICDE.2009.32
5. Bozkaya, T., Özsoyoglu, Z.M.: Indexing large metric spaces for similarity search queries. ACM Trans. Database Syst. **24**(3), 361–404 (1999)
6. Brin, S.: Near neighbor search in large metric spaces. In: VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11–15, 1995, Zurich, Switzerland, pp. 574–584 (1995)
7. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. (CSUR) **33**(3), 273–321 (2001)
8. Chen, L., Gao, Y., Li, X., Jensen, C.S., Chen, G.: Efficient metric indexing for similarity search. In: 31st (IEEE) International Conference on Data Engineering (ICDE), April 13–17, 2015, Seoul, South Korea, pp. 591–602 (2015). doi:10.1109/ICDE.2015.7113317
9. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25–29, 1997, Athens, Greece, pp. 426–435 (1997)
10. Dasgupta, S., Freund, Y.: Random projection trees for vector quantization. IEEE Trans. Inf. Theory **55**(7), 3229–3242 (2009)
11. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: D-index: distance searching index for metric data sets. Multimed. Tools Appl. **21**(1), 9–33 (2003)
12. Gao, J., Jagadish, H.V., Lu, W., Ooi, B.C.: DSH: data sensitive hashing for high-dimensional k-nnsearch. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, pp. 1127–1138 (2014)
13. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7–10, 1999, Edinburgh, Scotland, UK, pp. 518–529 (1999)
14. Goldstein, J., Ramakrishnan, R.: Contrast plots and p-sphere trees: space vs. time in nearest neighbour searches. In: Proceedings of 27th International Conference on Very Large Data Bases (VLDB), September 11–14, 2001, Roma, Italy, pp. 429–440 (2000)
15. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: Proceedings of 27th International Conference on Very Large Data Bases (VLDB), September 11–14, 2001, Roma, Italy, pp. 491–500 (2001)
16. Hellerstein, J.M., Naughton, J.F., Pfeffer, A.: Generalized search trees for database systems. In: VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11–15, 1995, Zurich, Switzerland, pp. 562–573 (1995)
17. Hjaltason, G.R., Samet, H.: Index-driven similarity search in metric spaces. ACM Trans. Database Syst. **28**(4), 517–580 (2003)
18. Jagadish, H.V., Ooi, B.C., Tan, K.-L., Yu, C., Zhang, R.: idistance: an adaptive B^+-tree based indexing method for nearest neighbor search. ACM Trans. Database Syst. **30**(2), 364–397 (2005)
19. Jr, C.T., Traina, A.J.M., Faloutsos, C., Seeger, B.: Fast indexing and visualization of metric data sets using slim-trees. IEEE Trans. Knowl. Data Eng. **14**(2), 244–260 (2002)
20. Kaufman, L., Rousseeuw, P.: Clustering by means of medoids. Statistical Data Analysis Based on the L1-Norm and Related Methods, pp. 405–416 (1987)
21. Koudas, N., Marathe, A., Srivastava, D.: Flexible string matching against large databases in practice. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB), August 31 - September 3, 2004, Toronto, Canada, pp. 1078–1086 (2004)
22. Lin, K.-I., Jagadish, H.V., Faloutsos, C.: The TV-tree: an index structure for high-dimensional data. VLDB J. **3**(4), 517–542 (1994)
23. Lu, W., Du, X., Hadjieleftheriou, M., Ooi, B.C.: Efficiently supporting edit distance based string similarity search using B^+-trees. IEEE Trans. Knowl. Data Eng. **26**(12), 2983–2996 (2014)
24. Lu, W., Shen, Y., Chen, S., Ooi, B.C.: Efficient processing of k nearest neighbor joins using mapreduce. PVLDB **5**(10), 1016–1027 (2012)
25. Novak, D., Batko, M., Zezula, P.: Metric index: an efficient and scalable solution for precise and approximate similarity search. Inf. Syst. **36**(4), 721–733 (2011)
26. Rong, C., Lu, W., Wang, X., Du, X., Chen, Y., Tung, A.K.H.: Efficient and scalable processing of string similarity join. IEEE Trans. Knowl. Data Eng. **25**(10), 2217–2230 (2013)
27. Uhlmann, J.K.: Satisfying general proximity/similarity queries with metric trees. Inf. Process. Lett. **40**(4), 175–179 (1991)
28. Venkateswaran, J., Lachwani, D., Kahveci, T., Jermaine, C.M.: Reference-based indexing of sequence databases. In: Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), September 12–15, 2006, Seoul, Korea, pp. 906–917 (2006)
29. Waterman, M.S.: Introduction to Computational Biology—Maps, Sequences, and Genomes: Interdisciplinary Statistics. CRC Press, Boca Raton (1995)
30. Winkler, W.E.: The state of record linkage and current research problems. Statistical Research Division, US Bureau of the Census (1999)
31. Yao, B., Li, F., Kumar, P.: K nearest neighbor queries and knn-joins in large relational databases (almost) for free. In: ICDE, pp. 4–15 (2010). doi:10.1109/ICDE.2010.5447837
32. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the Fourth Annual Symposium on Discrete Algorithms SODA, January 25–27, 1993, Austin, Texas, pp. 311–321 (1993)
33. Yoshitaka, A., Ichikawa, T.: A survey on content-based retrieval for multimedia databases. IEEE Trans. Knowl. Data Eng. **11**(1), 81–93 (1999)
34. Yu, C., Ooi, B.C., Tan, K.-L., Jagadish, H.V.: Indexing the distance: an efficient method to knn processing. In: Proceedings of 27th International Conference on Very Large Data Bases (VLDB), September 11–14, 2001, Roma, Italy, pp. 421–430 (2001)
35. Zhang, R., Kalnis, P., Ooi, B.C., Tan, K.-L.: Generalized multidimensional data mapping and query processing. ACM Trans. Database Syst. **30**(3), 661–697 (2005)
36. Zhang, Z., Hadjieleftheriou, M., Ooi, B.C., Srivastava, D.: Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In: Proceedings of the International Conference on Management of Data (SIGMOD), June 6–10, 2010, Indianapolis, Indiana, USA, pp. 915–926 (2010). doi:10.1145/1807167.1807266