Politecnico di Torino

Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA (COMPUTER ENGINEERING)
A.a. 2022/2023
Sessione di Laurea Luglio 2023

Similarity Join Queries

Techniques and Optimizations

Relatori:

Tania Cerquitelli
Stavros Sintos
Anastasios Sidiropoulos

Candidati:

Simone Zanella

ACKNOWLEDGMENTS

I want to thank my advisor, Stavros Sintos, his collaborator Xiao Hu and my Italian advisor, Tania Cerquitelli, for their time, effort and experience.

Finally I want to thank my family, whose support during these months and years has been fundamental to me.

SZ

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In this thesis we are going to propose efficient algorithms and data structures to handle similarity join queries over any number of constant relations in the dynamic setting with delay guarantees. We provide a lower bound complexity proof for the dynamic case, in which points are inserted and deleted after an initial preprocessing phase where all the needed data structures are created. We analyze special cases in which constraints allow us to efficiently answer a similarity join query exactly. Then, we design a grid-based approximation data structure for any dynamic similarity join query proving delay guarantees. A reduction to equi-joins is also provided. Finally an implementation of similarity join approximation algorithm with tests on different use cases is added at the end in appendices section.

CHAPTER 1

INTRODUCTION

Similarity join is a fundamental operator in databases [1] with many applications in data integration, data cleaning, bioinformatics, and pattern recognition. Similarity joins can be used in large databases to find similar products, or to identify relations between different points in a given dataset, which is a useful task for machine learning and data analysis purposes. There are multiple algorithms that have been proposed for similarity or the more general intersection join queries. In its most common form, we are given two sets of objects A, B and the problem is to report all pairs of objects $(a, b) \in A \times B$ such that $\phi(a, b) \leq r$, where $\phi(\cdot)$ is a distance function and r is a distance threshold. In databases, we assume two relations R_1, R_2 and the goal is to report pairs of tuples $(p_1, p_2) \in R_1 \times R_2$. The relations R_1, R_2 might contain any type of data, for example numerical attributes, geometric objects, strings, images etc.

In the era of big data, similarity join has become a crucial technique for effectively correlating data. This importance has led to its widespread adoption in various industries. For instance, Google employs both approximate and exact similarity join to identify near-duplicate web pages [2], mine query logs, and facilitate collaborative filtering [3]. Similarly, Microsoft has introduced the SSJoin primitive operator [4] to support similarity join, which has been utilized in the Data Debugger project [5].

In this project we assume that each relation contains a set of tuples, where each tuple has d numerical attributes. Equivalently, we can assume that each relation contains a set of points

in $\mathbb{R}^d$ and the goal is to find pairs of points from different relations that are within distance r. Notice that if the distance function ϕ is the ℓ_∞ (resp. ℓ_2) norm then an equivalent representation is that each relation contains a set of boxes (resp. balls), of side-length (resp. radius) $r/2$ and the goal is to find all pairs of boxes (resp. balls) from different relations that intersect. In the next sections we use both representations, using points and geometric objects, depending on what representation makes the problem easier to handle.

Most of previous works on designing algorithms for similarity join queries focus on two relations R_1, R_2. In this project we consider any number of constant relations that can be joined with any possible way. More specifically, we consider $k = O(1)$ relations $R_1, \ldots, R_k$ where each of them contains $O(n)$ points/tuples in $\mathbb{R}^d$. If two relations R_i, R_j are joined we use the notation $R_i \bowtie R_j$. Hence, in addition to the relations and the sets of points we are also given a join graph G_J that defines the similarity joins between the different relations. Handling multiple relations in similarity join queries is critical because modern databases often have complex data models that require joining more than two tables to retrieve meaningful results. For example, databases in E-commerce websites, Healthcare systems, and financial applications should join multiple relevant tables/relations.

While most of previous works consider the static case, we are interested in the *dynamic case* with *delay guarantees*. Due to streaming applications, and data integration from different sources, supporting dynamic updates is a key tool in database applications. For example, social media platforms such as Facebook and Twitter use dynamic updates to enable real-time updates of user feeds. We aim to construct a dynamic data structure that can be updated

efficiently under points' insertions or deletions. The data structure should support efficient enumeration of all similarity join results with delay guarantees. In general, delay guarantees in enumeration algorithms are important because they help ensure that the database is responsive to user queries. A delay guarantee specifies a maximum amount of time that an enumeration algorithm will take to return results. This allows users to specify a timeout period for their queries, so they do not have to wait indefinitely for a response. Formally speaking, for any function $g(\cdot)$, a delay guarantee of $O(g(n))$ is defined as a bound in the interval of time from the start of the enumeration to the first output, from one output to the next one and from last result to the end of the process.

Problem definition. Let $R_1, \ldots, R_k$ be $k = O(1)$ relations, and let $G_J(V, E)$ be the join graph. For each relation R_i we have a node $v_i \in V$ and two nodes are connected $(v_i, v_j) \in E$ if R_1 should be joined with R_2, i.e., $R_i \bowtie R_j$. Overall, G_J has $k = O(1)$ nodes and $O(k^2) = O(1)$ edges. Each R_i contains a set P_i of $O(n)$ points. Let $P = \bigcup_i P_i$. Let $\phi(\cdot, \cdot)$ be a distance function. In this project, we focus on ℓ_α norms as the distance function. We mostly consider the ℓ_2 and the ℓ_∞ norm. Finally, let $r > 0$ be a distance threshold. Most of the times we assume that $r = 1$.

The exact r-enumeration problem asks to construct a data structure to support the following operations.

- Update: When a new point is inserted in any of the k relations, update the data structure. Similarly, when a point is deleted from any of the k relations, update the data structure.

- Enumeration: When an enumeration query is received, all results $(p_1, \ldots, p_k) \in P_1 \times \ldots \times P_k$ should be enumerated with delay guarantees, such that for any $(v_i, v_j) \in E$, it should hold that $\phi(p_i, p_j) \leq r$.

We also study the approximation version of the same problem. In particular, if the enumeration returns all r-similarity join results along with some $(p'_1, \ldots, p'_k) \in P_1 \times \ldots \times P_k$ such that for any $(v_i, v_j) \in E$, $\phi(p_i, p_j) \leq (1 + \varepsilon)r$, then we say that the data structure supports ε-approximate r-enumeration. If r is clear from the context or $r = 1$, we skip it and we call them exact enumeration and ε-approximate enumeration.

Our Results. Using geometric algorithms along with techniques from database theory we show the following results:

- We show a conditional lower bound on the exact r-enumeration problem in 3 dimensions. In particular, we use the OuMV conjecture to reduce a modified version of the Online Matrix-Vector Multiplication problem to the exact r-enumeration problem for the line–3 similarity join $R_1 \bowtie R_2 \bowtie R_3$. The reduction is correct assuming that cos functions can be computed exactly.
- Even though the exact r-enumeration for the line–3 similarity join is expensive to solve, we show an efficient data structure that supports exact r–enumeration for line–3 similarity join in ℓ_∞ norm, when either R_1 or R_3 is static (for examples no updates are allowed in R_3). Furthermore, we show that if the points in R_2 have integer coordinates then there is an efficient dynamic data structure that supports exact r–enumeration for the line–3 similarity join in ℓ_∞ norm.

- We show a grid-based data structure of $O(n)$ space, that can be updated in $O(\varepsilon^{-kd} \log n)$ time and supports ε-approximate enumeration with $O(\varepsilon^{-kd} \log n)$ delay guarantee.
- Next we show how we can map general similarity join queries to a set of equi-join queries.
- We implement our new data structure that supports ε-approximate enumeration and run experiments on real and synthetic data testing the efficiency and efficacy of our methods.

Finally, we conclude with concluding remarks and future work.

CHAPTER 2

BACKGROUND

In this chapter we are going to explain some basic definitions, so that the reader is able to understand the thesis content.

Databases are fundamental entities used to store data on a long-term memory like an hard-disk. The way such entities store data must be well organized, so that looking for specific information inside it will take very little time. This process is called "data retrieval" and it is performed upon a request called "query". A query can be more or less complex based on the type of information we want to retrieve, the number of operations the system handling the database has to perform and other factors. Let's make an example to make the concept more clear. Let's assume we have a set of US cities whose coordinates in latitude and longitude are stored in our database like shown in Table I. An example of query would be: "Find all cities within 170 miles each other that form a triangle". Using the formula[1] to calculate the distance between 2 points on Earth, whose radius is 6371 km

$$\arccos\left(\sin(lat1) * \sin(lat2) + \cos(lat1) * \cos(lat2) * \cos(lon2 - lon1)\right) * 6371 \qquad (2.1)$$

it's easy to see that all possible solutions in our database are:

---

[1] `https://www.movable-type.co.uk/scripts/latlong.html`

TABLE I: US CITIES DATABASE EXAMPLE

Name	Latitude	Longitude
Los Angeles	34.0207305	-118.6919154
Tuscaloosa	33.2131155	-87.5368466
Chicago	41.8339037	-87.8720468
New York	40.6976637	-74.119764
Dallas	32.8208751	-96.8716362
Miami	25.7825453	-80.2994984
Boston	42.3144556	-71.0403236
Worcester	42.2754254	-71.8777776
Hartford	41.7657462	-72.7151063

- New York - Boston - Hartford
- New York - Boston - Worcester
- New York - Hartford - Worcester
- Boston - Worcester - Hartford

This kind of query, where a list of all the results is given as output, is called enumeration or reporting query. Queries, where just the existence of data satisfying a particular condition is asked, are called Boolean queries. Moreover counting queries are queries where the number of tuples satisfying a given condition is reported.

Let's add to our database some additional information like the state of each city, like shown in Table III. The query that we are going to perform on such database is similar to the previous one: "Find all cities belonging to different states within 170 miles each other that form a triangle". Given that Boston and Worcester are in the same state, the output is the following one:

- New York - Boston - Hartford
- New York - Hartford - Worcester

TABLE III: US CITIES DATABASE EXAMPLE WITH STATES

Name	State	Latitude	Longitude
Los Angeles	California	34.0207305	-118.6919154
Tuscaloosa	Alabama	33.2131155	-87.5368466
Chicago	Illinois	41.8339037	-87.8720468
New York	New York	40.6976637	-74.119764
Dallas	Texas	32.8208751	-96.8716362
Miami	Florida	25.7825453	-80.2994984
Boston	Massachusetts	42.3144556	-71.0403236
Worcester	Massachusetts	42.2754254	-71.8777776
Hartford	Connecticut	41.7657462	-72.7151063

This final example allows us to introduce similarity joins. A similarity join is a particular query where, given a dataset made of m different sets of d-dimensional points, the goal is to find points in each set such that a certain shape and distance between points belonging to different sets is satisfied. Moreover "join" is a common operation on databases and it could be of different types. For example in this thesis we will mention equi-joins, where the condition is expressed by equalities among attributes of different relations (sets of points), and interval joins, where data is represented by a set of segments and the condition is that they must overlap.
Another important idea is related to graphs. According to Wikipedia:

> "A graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called link or line)"

Graphs are going to be used to describe the similarity join shape.

CHAPTER 3

PREVIOUS WORK

The problem of reporting pairs or different shapes of points within a certain distance is a well known problem in database theory and computational geometry and it has different practical applications. Different papers have been written on the topic; paper [6], for example, shows two randomized solutions for a slightly modified version of our problem where just one set is taken into account, one metric (the Euclidean one) and pairs of points only. Both algorithms have delay guarantees, in particular the problem can be solved in $C(d)(n+k)\log n$, where d is data dimensionality, k the output size and n the input size. $C(d)$ is a constant dependant on d. One of the techniques used in the paper to handle high dimensional data (LSH) is used in the paper described in next section and extended to three points sets and triangles.

Another example taken from the literature is paper [7] where similarity joins are described according to 2013 implementation and used in relational databases. In particular the paper focuses mainly on string similarity.

The use of this kind of join is common for data cleaning purposes as highlighted in [8].

The thesis is based mainly on the work previously done on dynamic similarity joins[9]. Specifically the paper is about dynamic solutions for similarity joins using different metrics and techniques. To summarize, the paper shows both exact solutions for similarity joins and approximated solutions for joins which can't be solved efficiently and exactly. The metric used for

distances is fundamental because it changes the problem complexity; three metrics are analyzed in particular:

- l_∞ defined as $\min_{i\in[0,d)}|x_i - y_i|$
- l_1 defined as $\sum_{i=1}^{d}|x_i - y_i|$
- l_2 defined as $\sqrt{\sum_{i\in[0,d)}(x_i - y_i)^2}$

Fixing r and using l_1 and l_∞ metrics is possible to solve the problem efficiently and exactly, while the use of l_2 metric leads to efficient solutions for low dimensions only, that's why approximated solutions are developed for this metric, moreover for high dimensions an LSH approach is used.

3.1 Exact Enumeration

3.1.1 l_∞ metric

Let A and B be two point sets in $\mathbb{R}^d$ with $|A|+|B| = n$. For a point $p \in \mathbb{R}^d$, let $\mathscr{B}(p) = \{x \in \mathbb{R}^d | \|p-x\|_\infty \leq 1\}$ be the hypercube of side length 2. We wish to enumerate pairs $(a, b) \in A \times B$ such that $a \in \mathscr{B}(b)$.

Data structure. d-dimensional dynamic range tree on points of A with $O(n \log^{d-1} n)$ space and $O(n \log^d n)$ building time.

Update and Enumerate. Amortize update time for points in A: $O(\log^{2d} n)$, for points in B: $O(\log^{d+1} n)$. Enumerate: $O(\log^d n)$.

3.1.2 l_1 metric

l_1 metric is similar to l_∞ metric and it can be easily reduced to 2^d $(d+1)$-dimensional rectangle containment problems, obtaining the following results: $\tilde{O}(n)$ space, $\tilde{O}(n)$ building

time, $\tilde{O}(1)$ amortized update time and $\tilde{O}(1)$-delay enumeration. The procedure is described in detail in [9].

3.1.3 l_2 metric

In this case the similarity join problem has been reduced to halfspace-containment problem in $\mathbb{R}^{d+1}$.

The overall structure of the data structure is the same as the one used for l_1 metric, but instead of using a range tree, a dynamic partition tree is used for points in A. This structure of size $\tilde{O}(n)$ allows us to have $\tilde{O}(n^{2-\frac{1}{d+1}})$ building time, $\tilde{O}(n^{1-\frac{1}{d+1}})$ amortized updating time and $\tilde{O}(n^{1-\frac{1}{d+1}})$-delay enumeration of similarity join under the l_2 metric.

3.2 Approximate Enumeration

Approximate solutions work on any metrics and they report all pairs of $(a, b) \in A \times B$ with $\phi(a, b) \leq r$, along with (potentially) some pairs of (a', b') with $\phi(a', b') \leq r + \epsilon$, but no pair with $\phi(a, b) > r + \epsilon$.

The paper focuses on both variable distance threshold and fixed distance threshold. Focusing in particular on the last case we are going to introduce a grid-based data structure.

Data structure. The grid that we are going to define is used to split $\mathbb{R}^d$ space into cells of dimensions $\frac{\epsilon}{2\sqrt{d}}$ whose diameter is $\frac{\epsilon}{2}$. The distance between two cells c and c' is defined as: $\phi(c, c') = \min_{p \in c, q \in c'} \phi(p, q)$. Each grid cell stores two sets $A_c = A \cap c$ and $B_c = B \cap c$ and a value $m_c = \sum_{c':\phi(c,c') \leq 1} |B_{c'}|$ as the number of points in B that lie in a cell c' within distance 1 from cell c.

The only cells that are going to be stored are the non-empty ones, so the cells with one or more

elements. Moreover we introduce the notion of active cells; a cell is defined as active if and only if $A_c \neq \varnothing$ and $m_c > 0$. All active cells are stored in a balanced search tree, so that whether a cell c is stored can be answered in $O(\log n)$ time, similarly we store in a balanced search tree the set of non-empty cells.

Update. Let's assume that point $a \in A$ is inserted into cell c. If c is already non-empty then add a to A_c. Otherwise, we store c with $A_c = \{a\}$ and update m_c as follows. We visit each non-empty cell c' with $\phi(c, c') \leq 1$ and add $|B_{c'}|$ to m_c. If a point $a \in A$ is deleted the procedure is almost the same. Let's assume now point $b \in B$ is inserted into cell c. As in the previous case if c was empty we store it with $B_c = \{b\}$, if not we insert it into B_c and we visit all cells c' such that $\phi(c, c') \leq 1$ and we increase $m_{c'}$ by 1 and we store c into the active cells tree if c' turns from inactive to active. Similarly for b deletion. Overall the procedure is $\tilde{O}(\epsilon^{-d})$.

Enumeration. For each active cell c, we visit each non-empty cell c' within distance 1. If $B_{c'} \neq \varnothing$, we report all pairs of points in $A_c \times B_{c'}$. The procedure has $O(\epsilon^{-d} \log n)$ delay reporting one solution to the next one. For more details on the uniqueness of each pair of points in the enumeration phase see [9].

Another important paper, we are going to use later in "Reduction to equi-join" chapter, is [10]. In this section we are going to describe in more details the reduction they applies to intersection joins to make them equi joins. Given an intersection join, we build a segment tree on each common relation attribute and we encode each node as a bitstring: the empty string represents the root, the strings "0" and "1" represent the left and right child respectively, the string "00" and "01" represent the left and right child of the "0" respectively, and so on. Such

encoded nodes will be the tuples for the equi join queries we are going to create. Indeed each equi join query is meant to capture a different relation between nodes in the tree throught their bitstrings. Given two nodes n_1 and n_2, where n_1 is an ancestor of n_2, the bitstring for n_1 is then a prefix of that for n_2. To capture this relationship in a query, we can use a variable A_1 for n_1 bitstring and n_2 bitstring prefix and A_2 for the rest of n_2 bitstring. Each query will represent a single permutation among all the possibles taking into account all common variables. It's important to find such relations in the tree because of the major property of a segment tree: every interval stored in a node intersects all the intervals stored in children nodes. In order to better understand the reduction, we refer the reader to [10].

CHAPTER 4

LOWER BOUND COMPLEXITY

In this section we are going to prove, using the OuMv-conjecture, a lower bound for similarity joins. The OuMv conjecture is defined as follows. Given a Boolean matrix M of size $n \times n$ and a series of n pairs of vectors (u_i, v_i) provided dynamically, answering $u_i^T M v_i$ before getting (u_{i+1}, v_{i+1}) can't be done by any algorithm with total running time $O(n^{3-\epsilon})$, where $\epsilon \in \mathbb{R}^+$. The reduction can't be done in 2 dimensions given that we need, with at most n insertions or deletions for each pair, to select one row or column with just one point.

4.1 Reduction

The reduction is done in 3 dimensions. Each point $M[i][j]$ of the matrix, where $i, j \in \mathbb{N} \wedge i, j < n$, is mapped to a point on a cylinder surface with the following coordinates if and only if $M[i][j] = 1$.

$$\begin{cases} x = r \cos \frac{2\pi}{n} i \\ y = r \sin \frac{2\pi}{n} i \\ z = -\frac{\pi}{n^4} + \frac{2\pi}{n^5} j \end{cases} \tag{4.1}$$

r is the radius of the cylinder and $r \in \left(\sqrt{1 - \left(\frac{2\pi}{n^5}\right)^2}, 1 \right]$. Using the square root Taylor series we can write r without using the square root as $r \in \left[\frac{1}{2} + \frac{1}{2}\left(1 - \left(\frac{2\pi}{n^5}\right)^2\right), 1 \right]$. Such interval allows us to make sure, as we will describe in more details later, that each ball used for selecting the

matrix columns and placed in the center of the cylinder is not going to select other points in adjacent columns. Moreover the use of Taylor series allows us to get ride of the computational complexity of the square root and of any other function we are going to use the series on. Applying Taylor series on cosines and sines we get:

$$\begin{cases} x = r\left(1 - \frac{1}{2}\left(\frac{2\pi}{n}i\right)^2 + \left(\frac{2\pi}{n}i\right)^4\right) \\ y = r\left(\frac{2\pi}{n}i - \frac{1}{6}\left(\frac{2\pi}{n}i\right)^3 + \frac{1}{120}\left(\frac{2\pi}{n}i\right)^5\right) \\ z = -\frac{\pi}{n^4} + \frac{2\pi}{n^5}j \end{cases} \tag{4.2}$$

Vectors u and v are used for selecting rows and columns respectively. Starting with v, if $v[j] = 1$ a point in the center of the cylinder is placed, so that its coordinates are equal to

$$\begin{cases} x = 0 \\ y = 0 \\ z = -\frac{\pi}{n^4} + \frac{2\pi}{n^5}j \end{cases} \tag{4.3}$$

and the ball of radius 1 centered in the point inserted selects the j-th column.

In contrast if $u[i] = 1$ we place a point on a circle on the xy plane of radius $2\cos\frac{2\pi}{n}$ with the following coordinates:

$$\begin{cases} x = 2\cos\frac{2\pi}{n}\cos\frac{2\pi}{n}i \\ y = 2\cos\frac{2\pi}{n}\sin\frac{2\pi}{n}i \\ z = 0 \end{cases} \tag{4.4}$$

and using Taylor series:

$$\begin{cases} x = 2\left(1 - \frac{1}{2}\left(\frac{2\pi}{n}\right)^2 + \left(\frac{2\pi}{n}\right)^4\right)\left(1 - \frac{1}{2}\left(\frac{2\pi}{n}i\right)^2 + \left(\frac{2\pi}{n}i\right)^4\right) \\ y = 2\left(1 - \frac{1}{2}\left(\frac{2\pi}{n}\right)^2 + \left(\frac{2\pi}{n}\right)^4\right)\left(\frac{2\pi}{n}i - \frac{1}{6}\left(\frac{2\pi}{n}i\right)^3 + \frac{1}{120}\left(\frac{2\pi}{n}i\right)^5\right) \\ z = 0 \end{cases} \tag{4.5}$$

Such point is used for selecting the rows. A visual example with $n = 3$ of how the cylinder and the spheres are placed can be seen in Figure 1 and Figure 2. Whenever we apply Taylor series on cosines and sines we change i and j definitions so that we can have a better approximation of the value. Indeed we move the interval to $\left[-\frac{n}{2}, \frac{n}{2}\right)$ and $i, j \in \mathbb{Z}$. This is because all the series are centered in $x = 0$, so the approximation is almost perfect slightly around $x = 0$ and the further we go, the higher the approximation error will be.

Given that calculating a cosine or a sine value efficiently is always related with an error using a computer, we are limited in the value of n. Indeed for very high values of n each row can be so close that the approximation used for each cosine and sine values could lead the outer ball

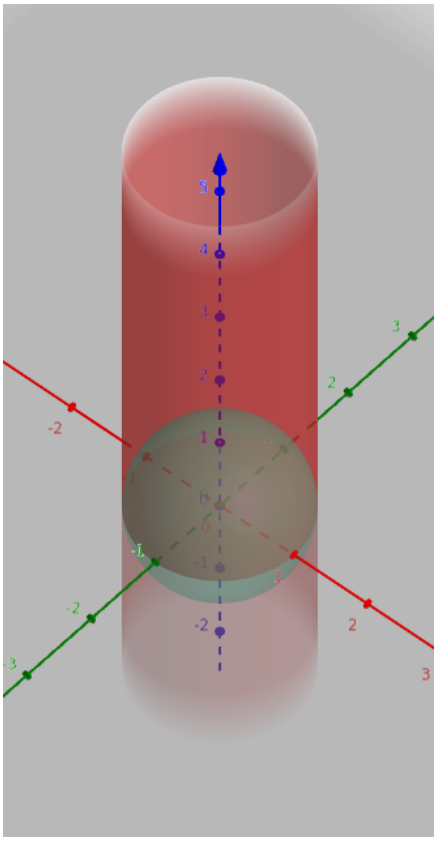


Figure 1: Cylinder and sphere used to select columns.

to intersect multiple rows.

In the end assuming exact values of cosines and sines and placing the point in the way just described allows us to state the following theorem.

Theorem 4.1.1 *Let be $\epsilon > 0$. If a dynamic algorithm with $t_u = n^{1-\epsilon}$ update time able to solve a similarity join in time $t_s = n^{2-\epsilon}$ on databases of size n exists, then OuMv can be solved in time $O(n^{3-\epsilon})$.*

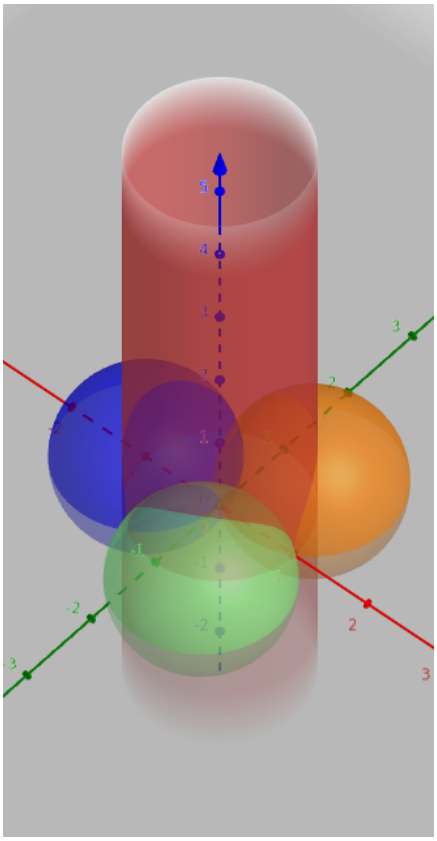
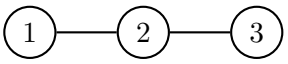


Figure 2: Cylinder and spheres used to select rows.

CHAPTER 5

EXACT SOLUTIONS FOR SPECIAL CASES OF LINE-3 SIMILARITY JOIN

In this chapter we are going to describe some special cases of line-3 similarity join in which can be solved efficiently. In all cases, let R_1, R_2, R_3 be three relations where each of them contains $O(n)$ tuples, or points in $\mathbb{R}^d$. We consider the line-3 join so we are interested in the schema $R_1 \bowtie R_2 \bowtie R_3$. The join graph looks like the next Figure:



Given a query, the goal is to report all results $(p, q, s) \in P_1 \times P_2 \times P_3$ such that $\phi(p, q) \leq r$, for a parameter r. In this section we consider that $\phi(p, q) = ||p - q||_\infty$.

5.1 Not everything is dynamic

We assume that either R_1 or R_3 is fixed. Without loss of generality, we consider that R_3 is fixed so we allow updates only in R_1 and R_2. We are going to map our problem to the problem of dynamic similarity join with delay guarantees for two relations. If $R'_1 \bowtie R'_2$ are the two relations to join, it is known from [9] how to solve the dynamic similarity join with delay guarantees problem efficiently.

Let DS be the data structure used in [9]. Let $insert(p, R'_j)$ be the procedure in [9] that inserts point p in R'_j (and DS) for $j \in \{1, 2\}$. Similarly, let $delete(p, R'_j)$ be the procedure in

[9] that deletes point p from R'_j (and DS) for $j \in \{1, 2\}$. Finally, let $query()$ be the procedure that reports all durable similarity joins in $R'_1 \bowtie R'_2$ with delay guarantees.

In our case, we start initializing the empty data structure DS. Furthermore, we construct a range tree $\mathcal{T}$ on the set of (fixed) points P_3.

Assume that a new point p is inserted in R_1. We simply call $insert(p, R_1)$. Next, assume that a new point p is deleted from R_1. We simply call $delete(p, R_1)$. Similarly, let p be a new point that is inserted in R_2. We define the square $\square_{p,r}$ with center p and radius r (so the side-length of $\square_{p,r}$ is $2r$. We run an *emptiness* query on $\mathcal{T}$ with query range $\square_{p,r}$. If $\square_{p,r} \cap P_3 = \varnothing$ we skip p and we end the update. Otherwise, we call $insert(p, R_2)$. Finally, if p is deleted from R_2, we simply call $delete(p, R_2)$. Using the results in [9], we have that the update operation takes $O(\log^{2d} n)$ amortized time.

After number of updates, assume that we receive a query. We run $query()$. For each pair $(p, q) \in P_1 \times P_2$ we receive, we define the square $\square_{p,r}$ and run a reporting query on $\mathcal{T}$. For each $s \in P_3$ is returned, we report (p, q, s). The $query()$ procedure in [9] has $O(\log^d n)$ delay guarantee. We also need to run another reporting query on $\mathcal{T}$, so the overall delay is $O(\log^{2d} n)$.

Putting everything together, we have:

Theorem 5.1.1 *Let R_1, R_2, R_3 be three relations and P_1, P_2, P_3 their corresponding sets of points in $\mathbb{R}^d$ with $|P_i| = O(n)$ for each $i \leq 3$. Let R_3 be a fixed relation meaning that no insertions or deletions are allowed in R_3. Let r be a given distance threshold. There exists a data structure of $O(n \log^{d-1} n)$ space that can be updated in $O(\log^{2d} n)$ amortized time, while supporting exact r-enumeration of the similarity line–3 join with $O(\log^{2d} n)$ delay.*

5.2 Integer coordinates

In this section we assume that the coordinates of the points in R_2 are integers.

We construct the grid over $\mathbb{R}^d$ where each grid cell has side-length 1. In particular, we only store in an AVL tree the grid vertices that contain at least one point from P_2. For each grid vertex v we store a hash-map or a search binary tree T_v that stores the points from R_2 that lie on v. Let $v.c_1$ be a counter that stores the number of points from P_1 within distance r from v. Similarly, let $v.c_3$ be a counter that stores the number of points from P_3 within distance r from v. A vertex v in the grid is active if and only if i) there is a point $p_2 \in P_2$ such that $p_2 \in T_v$, ii)$v.c_1 > 0$, and iii) $v.c_3 > 0$. Let v_b be a boolean variable v_b which is 1 if v is active and 0 if it is not active. Let T_a be a search binary tree that stores the active grid vertices. Let T_1 and T_3 be two dynamic range trees built on P_1 and P_3, respectively. The overall data structure can be constructed has space $O(n \log^{d-1} n)$.

We consider all the different cases. For any point p, let $\square_{p,x}$ be a box with center p and radius x.

If a point p_1 is inserted in P_1, we insert it in T_1, and we use the AVL tree to identify all grid vertices in $\square_{p_1,r}$, called V_{p_1}. For each $v \in V_{p_1}$, we increase $v.c_1 \leftarrow v.c_1 + 1$. If $v_b = 0$ and $v.c_2 > 0$ we set $v_b = 1$ and we insert v in T_a. Next, assume that we remove a point $p_1 \in P_1$. We remove it from T_1 and we use the AVL tree to identify all grid vertices in $\square_{p_1,r}$, called V_{p_1}. For each $v \in V_{p_1}$, we decrease $v.c_1 \leftarrow v.c_1 - 1$. If $v_b = 1$ and $v.c_1 = 0$ we set $v_b = 0$ and we remove v from T_a. Similarly, we can handle insertion and deletion from P_3.

If point p_2 is inserted in P_2, we use the AVL tree and find its grid vertex. If it does not exist, we create it. Let v be its grid vertex. We add p_2 in T_v. If T_v contains only one point (the new point p_2 we inserted) we run a count query in T_1 with query range $\square_{v,r}$. Let c_1 be the result of the count query. We update $v.c_1 = c_1$. Furthermore, we run a count query in T_3 with query range $\square_{v,r}$. Let c_3 be the result of the count query. We update $v.c_3 = c_3$. If $v_b = 0$ and $v.c_1 > 0$, $v.c_2 > 0$ we set $v_b = 1$ and we insert v in T_a. Finally, if we delete point p_2 from P_2, we use the AVL tree and find its grid vertex v. We remove p_2 from T_v. If $T_v = \varnothing$, we remove v from T_a and we remove v from the AVL tree.

Next, we describe the enumeration procedure. For each $v \in T_a$, we run a reporting query in T_1 and a reporting query in T_3 with query range $\square_{v,r}$. Let $A = P_1 \cap \square_{v,r}$, $B = T_v$, $C = P_3 \cap \square_{v,r}$ found by T_1, T_v, T_3, respectively. In particular, we do not report the results A or C when we run the reporting query. Instead we find the canonical nodes in the trees. Then, we report $A \times B \times C$.

Even though the algorithm we describe supports exact r–enumeration, the correctness of all the procedure follow using similar arguments with our new grid-based approximation algorithm in the next section.

In order to handle updates in P_1, P_3 we need $O(r^d + \log^d n)$ time because we update the dynamic tree T_1 and the size of V_{p_1} is at most $O(r^d)$. In order to handle updates in P_2 we need $O(\log^d n)$ time because we run two queries in range trees T_1 and T_3. All operations in the AVL tree and T_a, T_v can be executed in $O(\log n)$. Finally, the enumeration procedure has $O(\log^d n)$ delay because it runs range queries in T_1, T_3. These procedures dominate the time.

Overall, putting everything together, we get the following result.

Theorem 5.2.1 *Let R_1, R_2, R_3 be three relations and P_1, P_2, P_3 their corresponding sets of points in $\mathbb{R}^d$ with $|P_i| = O(n)$ for each $i \leq 3$. Assume that all coordinates of all points in P_2 are integers. Let r be a given distance threshold. There exists a data structure of $O(n \log^{d-1} n)$ space that can be updated in $O(r^d + \log^d n)$ time, while supporting exact r-enumeration of the similarity line–3 join with $O(\log^d n)$ delay.*

CHAPTER 6

ANY SHAPE SIMILARITY JOIN

Using the same approach based on the grid method explained before we can solve any kind of similarity join, no matter the number of relations and the shape of the join. In order to achieve such result we just need to slightly modify the update process and some definitions.

6.1 Approximation algorithm

Let $R_1, \ldots, R_k$ be $k = O(1)$ relations, and let $G_J(V, E)$ be the join graph with k nodes and $O(k^2) = O(1)$ edges. Each R_i contains a set P_i of $O(n)$ points. Let $P = \bigcup_i P_i$. The goal is to report all similarity join results with respect to G_J with $O(\varepsilon^{-kd} \log n)$ delay. In particular, report $p_i \in P_i$ for each i if and only if for every $(v_j, v_h) \in E$, $||p_j - p_h|| \leq 1$. If for some (or all) edges $(v_j, v_h) \in E$ it holds that $1 \leq ||p_j - p_h|| \leq 1+\varepsilon$ then we say that we have an ε-approximate enumeration of similarity join G_J. The goal is to construct an index that supports the following operations.

- We insert a point p in R_i (for any i) in $O(\varepsilon^{-kd} \log n)$ time.
- We remove a point $p \in R_i$ (for any i) in $O(\varepsilon^{-kd} \log n)$ time.
- Supports ε-approximate enumeration of similarity join G_J with $O(\varepsilon^{-kd} \log n)$ delay.

6.1.1 Index description.

We construct a grid G in R^d such that the grid cell diagonal is $\varepsilon/2$. We place the grid cells in a AVL tree T. Without loss of generality, let R_1 be the *key* relation. For each cell $c \in G$ and

each $i \leq k$ we store $P_i^c = P_i \cap c$, $m_i^c = |P_i^c|$. For each cell $c \in G$ s.t $m_1^c > 0$, we store an integer value m_c which represents the number of similarity join results with respect to G_J with c as the *key* cell. Each cell c s.t. $m_c > 0$ is also stored in another AVL tree called active cells tree.

6.1.2 Point insertion.

Let p_j be a new point that we insert in P_j. Using the AVL tree we check whether the grid cell that p_j belongs to, exists. If not, we construct it. Let $c \in G$ be the cell such that $p_j \in c$. We add $p_j \in P_j^c$, we set $m_j^c = m_j^c + 1$ and we update m_c accordingly.

In this paragraph, we describe a procedure to find the cells that contain a point from P_1 that are affected by the insertion of p_j. We start from the cell c of G. In a high level, the algorithm is a recursive approach running a BFS to the graph G_J starting from node v_j. Each node $v \in V$ is associated with a list of cells from G which represents all possible cells assignments for node v in the current iteration. Initially each v_i is associated with $\varnothing$, except v_j which is associated with c. Let each cell c contained in v_j list be c'. Assign c' in the final solution fs. Let $(v_j, v_h) \in E$ and the algorithm has visited v_h in a previous iteration and has assigned a cell $\hat{c}$ to v_h in fs[1]. We check whether $||c' - \hat{c}|| \leq 1$. If not, then we skip cell c' from v_h list, erasing any previous assignment to neighbor nodes both in fs and neighbors' lists. On the other hand, let $(v_j, v_h) \in E$ and the algorithm has not visited v_h. We consider all cells in G around c' within distance 1, i.e. $||c' - \bar{c}|| \leq 1$. For each such cell $\bar{c}$ we check whether $m_h^{\bar{c}} > 0$. If yes, we insert

[1]Notice that we do not simply run a BFS. Instead we visit the nodes of the graph in a BFS manner assigning a cell to each node. So we might visit a node multiple times checking on whether or not the assigned cell is valid.

$\bar{c}$ into v_h list. If v_h list is empty, we skip cell c' from v_j list, erasing any previous assignment to neighbor nodes both in fs and neighbors' lists. If v_h list is not empty and we were able to assign to each non-visited v_j neighbor some cells, then we push each non-visited neighbor in the queue, we mark it as visited and we recur. When the queue is empty we know for sure that a valid solution has been found, so we push fs into a list of solutions.

Let $c_1, \ldots, c_k$ be an element of the final list of solutions. Without loss of generality assume that c_i is assigned to v_i. We set $m_c^{c_1} = \prod_{i=1}^{k} m_i$ and, given that at this point at least one solution exists, we are sure that $m_c^{c_1} > 0$, so we add c_1 to active cells tree.

For each node we consider $O(\varepsilon^{-d})$ cells around it. There are k nodes so in total we visit $O(\varepsilon^{-kd})$ cells. Each time we need $O(\log n)$ time to check if a cell exists in grid G. The insertion is executed in $O(\varepsilon^{-kd} \log n)$ time.

6.1.3 Point deletion.

The deletion procedure is similar to the insertion but there are a couple of key differences. When we delete a point $p_j \in P_j^c$ from cell c we update $P_j^c = P_j^c \setminus \{p_j\}$ and $m_j^c = m_j^c - 1$. Next we have to update m_c of cells c_1 that belong to some similarity join result. Following the same procedure described for the insertion case we find $c_1, \ldots, c_k$, we set $m_c^{c_1} = \prod_{i=1}^{k} m_i$. If $m_c^{c_1}$ is exactly zero then we delete c_1 from active cells tree. Finally, if c is empty we delete it from cells tree. The deletion is executed in $O(\varepsilon^{-kd} \log n)$ time.

Algorithm 1: Point Insertion Algorithm

input: similarity join graph G_J, point p_j
Set $c \in G$ s.t. $p_j \in c$
set $P_j^c = P_j^c \cup \{p_j\}$
$m_j^c = m_j^c + 1$
Let Q be a queue
Let v_j be the vertex associated to p_j relation
foreach *Vertex v in* G_J **do**
  Assign an empty list of cells
end
Set v_j list to $[c]$
Mark v_j as visited
Push v_j into Q
Point_Insertion_Algorithm_Recursive(Q, G_J, all vertices lists)
if *a valid solution has been found* **then**
  Set $m_c^{c_1} = \prod_{i=1}^{k} m_i$
  Add c_1 to active cells tree
end

6.1.4 Query procedure

We visit each cell c of active cells tree. For each such cell, we run an algorithm related to the point insertion algorithm. In particular, we mimic the algorithm assuming that we added the first point P_q^c in c (given that the other point pre-existed). While the BFS is running, let $c_1, \ldots, c_k$ be the cells associated with the k nodes. We report $P_1^{c_1} \times \ldots \times P_k^{c_k}$. It is straightforward to see that the described procedure executes an ε-approximate enumeration of G_J.

Proof. Let's assume that a point $p \in P_1$ lies on the left edge of cell c and that the similarity join distance is 1. Be c' a cell whose distance from c left edge is 1. In this case such cell, as well as all points stored in it, will be taken into account in the enumeration phase. Let's assume

Algorithm 2: Point Insertion Algorithm Recursive

input: Queue Q, similarity join graph G_J, all possible cell assignments by vertex
if Q *is empty* **then**
  Save current solution
end
Pop Q into v_j
foreach *cell* $c' \in v_j$ *list* **do**
  Assign c' to final solution
  foreach *neighbor* v_h *of* v_j **do**
    if v_h *visited* **then**
      if v_h *has a cell assigned in the final solution* **then**
        Let $\bar{c}$ be the cell assigned to v_h in the final solution
        if $||c' - \bar{c}|| > 1$ **then**
          Erase all previously assigned non-visited node lists
          Skip c'
        end
      end
    end
    else
      foreach *cell* $\bar{c}$ *s.t.* $m_h^{\bar{c}} > 0 \wedge ||c' - \bar{c}|| \leq 1$ **do**
        Add $\bar{c}$ to v_h list
      end
      if v_h *list is empty* **then**
        Erase all previously assigned non-visited node lists
        Skip c'
      end
    end
  end
  foreach *non-visited neighbor* v_h **do**
    Mark v_h as visited
    Push v_h into Q
  end
  Point_Insertion_Algorithm_Recursive(Q, G_J, all vertices lists)
  foreach *non-visited neighbor* v_h **do**
    Mark v_h as not visited
    Pop Q
    Erase v_h list
  end
  Remove c' from final solution
end

that point $p_1 \in c'$ lies on the right edge, it's easy to see that $||p - p_1|| \leq (1 + \epsilon)$ and that every other point p_2 with $||p - p_2|| > (1 + \epsilon)$ will be stored in cells nearby, so it won't be reported.

Following the analysis we had before, we get that the delay guarantee is $O(\varepsilon^{-kd} \log n)$.

Theorem 6.1.1 *Let $R_1, \ldots, R_k$ be $k = O(1)$ relations with join graph G_J, and $P_1, \ldots, P_k$ their corresponding sets of points in $\mathbb{R}^d$ with $|P_i| = O(n)$ for each $i \leq k$. There exists a data structure of $O(n)$ space that can be updated in $O(\varepsilon^{-kd} \log n)$ time, while supporting ε-approximate enumeration of the similarity join G_J with $O(\varepsilon^{-kd} \log n)$ delay.*

CHAPTER 7

REDUCTION TO EQUI-JOIN

Every similarity join can be reduced to an equi join through a previous reduction to intersection joins. In this chapter we are going to show how the reduction works.

7.1 Static case.

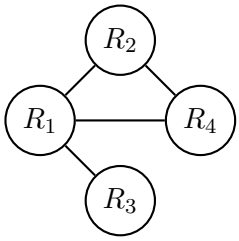
The reduction from an intersection join to an equi join has already been studied in paper [10]. Overall the process relies on the construction of segments trees on common interval attributes, encoding each tree node using bitstrings which will be stored in the final equi join relations.

Specifically given a shape for the similarity join, encoded as a graph with $|V|$ vertices and $|E|$ edges, and d the data dimension, running algorithm 3 allows us to create all needed relations where $[A_1], \ldots, [A_d]$ is the set of intervals delimited by point p coordinates $\pm\frac{similarityJoinDistance}{2}$ for each dimension. Having half of the similarity join distance is fundamental because the concept of intersecting intervals has to be mapped to the concept of centers distance being at most the similarity join distance. In this way we are sure that extreme cases like two squares sharing an edge are covered by our reduction.

Algorithm 3: FromSimilarityToIntersection

```
input: graph G
for edge e ∈ E do
 | Assign a new set of [A_1], ..., [A_d] to e
end
i = 0
for vertex v ∈ V do
 | Create relation R_i
 | for edge e s.t. v ∈ e do
 |  | Add to R_i as attributes all [A_1], ..., [A_d] assigned to e
 | end
 | i += 1
end
return All relations R_i
```

To better explain the reduction we describe the following example, where, for simplicity, the relations' names are already written in the center of each vertex, even if for similarity joins we don't exactly have relations, but d-dimensional points sets.



Assuming $d = 2$ algorithm 3 would return:

$$R_1\left([A_1], [A_2], [A_5], [A_6], [A_7], [A_8]\right), R_2\left([A_1], [A_2], [A_3], [A_4]\right), R_3\left([A_7], [A_8]\right)$$

$$R_4\left([A_3], [A_4], [A_5], [A_6]\right)$$

Using paper [10] we can build the corresponding equi join. The way the reduction is done allows us to state the following theorem.

Theorem 7.1.1 *Given a graph G representing the similarity join shape, if G is acyclic, the intersection join obtained using algorithm 3 with hypergraph $\mathcal{H}_i$ will be ι-acyclic, so the final equi join hypergraph $\mathcal{H}_e$ will be α-acyclic as well.*

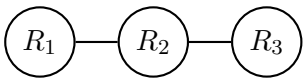
Moreover if $d \geq 2$ and the number of points' sets is greater or equal than two for similarity joins, then the reduction will lead to non q-hierarchical queries. According to paper [11]:

> "
>
> A conjunctive query ϕ is q-hierarchical if for any two variables $x, y \in \text{vars}(\phi)$ the following conditions are satisfied:
>
> - $\text{atoms}(x) \subseteq \text{atoms}(y)$ or $\text{atoms}(x) \supseteq \text{atoms}(y)$ or $\text{atoms}(x) \cap \text{atoms}(y) = \varnothing$
> - if $\text{atoms}(x) \subsetneq \text{atoms}(y)$ and $x \in \text{free}(\phi)$, then $y \in \text{free}(\phi)$
>
> "

Where $\text{free}(\phi)$ is the set of variables that appears in all atoms and that are not used as projection variables. Given that variables can be easily considered intervals, such definition can be extended to intersection joins too. Moreover the reduction to equi join doesn't change the overall query structure and the attributes are just substituted with different $[A_1], \ldots, [A_d]$ according to the permutations, a more detailed proof will follow the theorem.

Theorem 7.1.2 *Given a graph G representing the similarity join shape with $d \geq 2$ and the number of points' sets > 2, the intersection join obtained using algorithm 3 will be non q-hierarchical, so the final equi join will be non q-hierarchical as well.*

In particular let's assume $d = 2$, the number of relations equal to 3 and the following shape:



As we did before, running algorithm 3 we get:

$$R_1\left([A_1],[A_2]\right), R_2\left([A_1],[A_2],[A_3],[A_4]\right), R_3\left([A_3],[A_4]\right)$$

It's easy to see how condition one of q-hierarchical queries doesn't hold in this case.

Moreover the full reduction to equi join queries leads to 16 different permutations that are reported with a little change of variable names for readability:

$$\tilde{Q}_1 = \tilde{R}_1\left(A_{11}, B_{11}\right), \tilde{R}_2\left(A_{11}, A_{12}, B_{11}, B_{12}, A_{21}, B_{21}\right), \tilde{R}_3\left(A_{21}, A_{22}, B_{21}, B_{22}\right)$$

$$\tilde{Q}_2 = \tilde{R}_1\left(A_{11}, B_{11}\right), \tilde{R}_2\left(A_{11}, A_{12}, B_{11}, B_{12}, A_{21}, B_{21}, B_{22}\right), \tilde{R}_3\left(A_{21}, A_{22}, B_{21}\right)$$

$$\tilde{Q}_3 = \tilde{R}_1\left(A_{11}, B_{11}\right), \tilde{R}_2\left(A_{11}, A_{12}, B_{11}, B_{12}, A_{21}, A_{22}, B_{21}\right), \tilde{R}_3\left(A_{21}, B_{21}, B_{22}\right)$$

$$\tilde{Q}_4 = \tilde{R}_1\left(A_{11}\right), \tilde{R}_2\left(A_{11}, A_{12}, B_{11}, B_{12}, A_{21}, A_{22}, B_{21}, B_{22}\right), \tilde{R}_3\left(A_{21}, B_{21}\right)$$

$$\tilde{Q}_5 = \tilde{R}_1\left(A_{11}, B_{11}, B_{12}\right), \tilde{R}_2\left(A_{11}, A_{12}, B_{11}, A_{21}, B_{21}\right), \tilde{R}_3\left(A_{21}, A_{22}, B_{21}, B_{22}\right)$$

$$\tilde{Q}_6 = \tilde{R}_1\left(A_{11}, B_{11}, B_{12}\right), \tilde{R}_2\left(A_{11}, A_{12}, B_{11}, A_{21}, B_{21}, B_{22}\right), \tilde{R}_3\left(A_{21}, A_{22}, B_{21}\right)$$

$$\tilde{Q}_7 = \tilde{R}_1\left(A_{11}, B_{11}, B_{12}\right), \tilde{R}_2\left(A_{11}, A_{12}, B_{11}, A_{21}, A_{22}, B_{21}\right), \tilde{R}_3\left(A_{21}, B_{21}, B_{22}\right)$$

$$\tilde{Q}_8 = \tilde{R}_1\left(A_{11}, B_{11}, B_{12}\right), \tilde{R}_2\left(A_{11}, A_{12}, B_{11}, A_{21}, A_{22}, B_{21}, B_{22}\right), \tilde{R}_3\left(A_{21}, B_{21}\right)$$

$$\tilde{Q}_9 = \tilde{R}_1\left(A_{11}, A_{12}, B_{11}\right), \tilde{R}_2\left(A_{11}, B_{11}, B_{12}, A_{21}, B_{21}\right), \tilde{R}_3\left(A_{21}, A_{22}, B_{21}, B_{22}\right)$$

$$\tilde{Q}_{10} = \tilde{R}_1\left(A_{11}, A_{12}, B_{11}\right), \tilde{R}_2\left(A_{11}, B_{11}, B_{12}, A_{21}, B_{21}, B_{22}\right), \tilde{R}_3\left(A_{21}, A_{22}, B_{21}\right)$$

$$\tilde{Q}_{11} = \tilde{R}_1\left(A_{11}, A_{12}, B_{11}\right), \tilde{R}_2\left(A_{11}, B_{11}, B_{12}, A_{21}, A_{22}, B_{21}\right), \tilde{R}_3\left(A_{21}, B_{21}, B_{22}\right)$$

$$\tilde{Q}_{12} = \tilde{R}_1\left(A_{11}, A_{12}, B_{11}\right), \tilde{R}_2\left(A_{11}, B_{11}, B_{12}, A_{21}, A_{22}, B_{21}, B_{22}\right), \tilde{R}_3\left(A_{21}, B_{21}\right)$$

$$\tilde{Q}_{13} = \tilde{R}_1\left(A_{11}, A_{12}, B_{11}, B_{12}\right), \tilde{R}_2\left(A_{11}, B_{11}, A_{21}, B_{21}\right), \tilde{R}_3\left(A_{21}, A_{22}, B_{21}, B_{22}\right)$$

$$\tilde{Q}_{14} = \tilde{R}_1\left(A_{11}, A_{12}, B_{11}, B_{12}\right), \tilde{R}_2\left(A_{11}, B_{11}, A_{21}, B_{21}, B_{22}\right), \tilde{R}_3\left(A_{21}, A_{22}, B_{21}\right)$$

$$\tilde{Q}_{15} = \tilde{R}_1\left(A_{11}, A_{12}, B_{11}, B_{12}\right), \tilde{R}_2\left(A_{11}, B_{11}, A_{21}, A_{22}, B_{21}\right), \tilde{R}_3\left(A_{21}, B_{21}, B_{22}\right)$$

$$\tilde{Q}_{16} = \tilde{R}_1\left(A_{11}, A_{12}, B_{11}, B_{12}\right), \tilde{R}_2\left(A_{11}, B_{11}, A_{21}, A_{22}, B_{21}, B_{22}\right), \tilde{R}_3\left(A_{21}, B_{21}\right)$$

We can easily verify that each query is non q-hierarchical. This implies that the overall reduction leads to non q-hierarchical queries and, given that the example above is based on the simplest version of similarity join with $d \geq 2$ and the number of points' sets > 2 and that it can be found as a subcase in any other more complicated version, the previous theorem is proved.

7.2 Dynamic case.

In this section we are going to analyze the impact that inserting a point into or deleting a point from the original similarity join dataset have on the reduction. Assuming data dimensionality d, inserting a point in any points' set means inserting in the intersection join obtained using algorithm 3 a new tuple made of the intervals described in the previous section. According to [10], we have then to insert such intervals into some segments trees in order to get the corresponding bitstring, which will be inserted in the equi join database $\tilde{D}_{[X]}$, using the cited paper notation.

CHAPTER 8

EXPERIMENTS

8.1 Triangle similarity join - ϵ approximation implementation

We provide an implementation of the grid method for similarity join approximation described in [9] applied to triangle shapes and mainly two dimensions. All code can be found in the following link: https://github.com/Simone99/Any-shape-similarity-join-approx

The repository is organized as described in Table V.

Different experiments have been made using different datasets uniformly and randomly generated with values between 0 and 10 and all of them have been run on a 11th Gen Intel® Core™ i7-11390H machine with 16GB of RAM.

The parameters which have been modified during the tests are:

- R. The similarity join distance.
- #dimensions. Data dimensionality.
- ϵ. Distance approximation.

The abbreviations used in Table VII headers are:

- # for the number of dimensions.
- GCT for grid creation time. It includes initialization time as well.
- QT for query time.

TABLE V: GITHUB REPOSITORY STRUCTURE

AvlTree.hpp	Contains all the code needed to implement a self-balancing binary search tree which is used to store all non-empty and active cells
Database.cpp/Database.hpp	Contains all the code to randomly generate or read from a file data points. Each point is stored in different sets which will be used to create the grid and answer the queries
Grid.cpp/Grid.hpp	Contains all the code to create the grid with ϵ radius cells. Moreover it contains all the logic related to dynamic database updates, covering insertion and deletion with m_c update.
main.cpp	Contains the overall program logic. It creates the database, the grid and it performs measurements on the algorithm performance in different scenarios.
Makefile	File used to compile the program successfully.
plot_points.py	Python script used to visualize the query result.

- RT for real triangles, so triangles with edges long less than or at most the similarity join distance.
- AT for approximated triangles, so triangles with edges long more than the similarity join distance.
- AR for approximation ratio.

TABLE VII: TEST RESULTS FOR TRIANGLE SIMILARITY JOIN

R	#	ϵ	GCT[s]	QT[s]	RT	AT	AR
3000 points							
1.5	2	0.1	10.159	19.344	2419786	53876	1.02226
1.0	2	0.1	3.716	5.85	508706	17738	1.03487
0.5	2	0.1	1.948	2.134	32597	2600	1.07976
2.0	2	0.1	26.775	53.949	7143080	118996	1.01666
4.0	2	0.1	276.562	610.736	85937029	629818	1.00733
1.5	2	0.1	10.343	20.004	2419786	53876	1.02226
1.5	2	0.35	13.489	30.955	2419786	1478590	1.61104
1.5	2	0.065	10.345	19.566	2419786	17457	1.00721
1.5	2	1.0	3.253	67.073	2419786	16789486	7.93842
1.5	2	0.01	10.082	19.104	2419786	39	1.00002
1.5	7	0.1	1.036	0.0	0	0	1.0

1.5	3	0.1	1.989	2.284	71797	33	1.00046
1.5	4	0.1	1.602	1.049	1742	0	1.0
1.5	2	0.1	11.161	20.062	2394802	59500	1.02485
300 points							
1.5	2	0.1	0.033	0.044	2628	11	1.00419
1.0	2	0.1	0.026	0.023	533	7	1.01313
0.5	2	0.1	0.02	0.008	30	0	1.0
2.0	2	0.1	0.055	0.113	7783	11	1.00141
4.0	2	0.1	0.301	0.58	91166	155	1.0017
1.5	2	0.1	0.038	0.042	2628	11	1.00419
1.5	2	0.35	0.038	0.049	2628	291	1.11073
1.5	2	0.065	0.036	0.042	2628	3	1.00114
1.5	2	1.0	0.061	0.1	2628	6061	3.30632
1.5	2	0.01	0.035	0.04	2628	0	1.0
1.5	7	0.1	0.015	0.0	0	0	1.0
1.5	3	0.1	0.022	0.005	60	0	1.0
1.5	4	0.1	0.016	0.0	0	0	1.0
1.5	2	0.1	0.032	0.039	2025	6	1.00296
150 points							
1.5	2	0.1	0.007	0.008	282	4	1.01418

1.0	2	0.1	0.008	0.003	63	0	1.0
0.5	2	0.1	0.006	0.0	7	0	1.0
2.0	2	0.1	0.012	0.013	858	0	1.0
4.0	2	0.1	0.046	0.083	10029	0	1.0
1.5	2	0.1	0.007	0.007	282	4	1.01418
1.5	2	0.35	0.007	0.009	282	19	1.06738
1.5	2	0.065	0.007	0.008	282	0	1.0
1.5	2	1.0	0.012	0.014	282	401	2.42199
1.5	2	0.01	0.007	0.008	282	0	1.0
1.5	7	0.1	0.004	0.0	0	0	1.0
1.5	3	0.1	0.006	0.0	3	0	1.0
1.5	4	0.1	0.004	0.0	1	0	1.0
1.5	2	0.1	0.009	0.006	288	0	1.0

As we can see from Table VII the smaller the radius R, the less time is needed in order to answer the query. Given that we used the same dataset to run all the experiments with same data dimensionality, a ball with radius R centered in the same point will contain less points if the radius R decreases in size, so the experiment confirms the theoretical results: enumeration complexity is dependant on the output size. Same reasoning applies to ϵ, a larger ϵ can increase the output size and accordingly the running time. Finally, in each case taken into account in this paper d was a constant, so some experiments have been made to show how data dimension-

ality affects query time. In general if d is not a constant, the query time should increase, but the experiments highlight a different phenomenon: Curse of dimensionality. According to this phenomenon, as data dimensionality increases, data becomes more sparse, which means that is more difficult to find points at a distance less than R, so less points have to be reported. In some cases, when the dataset is small, the query time is 0s, so no results have to be reported or the number of triangles to report is very tiny. Moreover we can see how the choice of ϵ is going to influence the number of approximated triangles, indeed the worst approximation ratio is obtained by setting ϵ to 1. Unfortunately the approximation algorithm described in [9], as the tests confirm, doesn't provide a bounded approximation ratio.

In Figure 3 we can see an example of output of a triangle similarity join on a small dataset.

8.2 Any shape similarity join - ϵ approximation implementation

The code showed in previous chapter has been extended to execute and run all tests with different shapes according to the algorithm explained in this paper. All code is stored in branch "general_shape" whose structure is equal to the previous one except for minor changes.

An undirected graph is used to describe the similarity join shape, in particular "input_graph.txt" has the following structure: first line is the number of vertices, second line the number of edges and all following lines represent a list of edges.

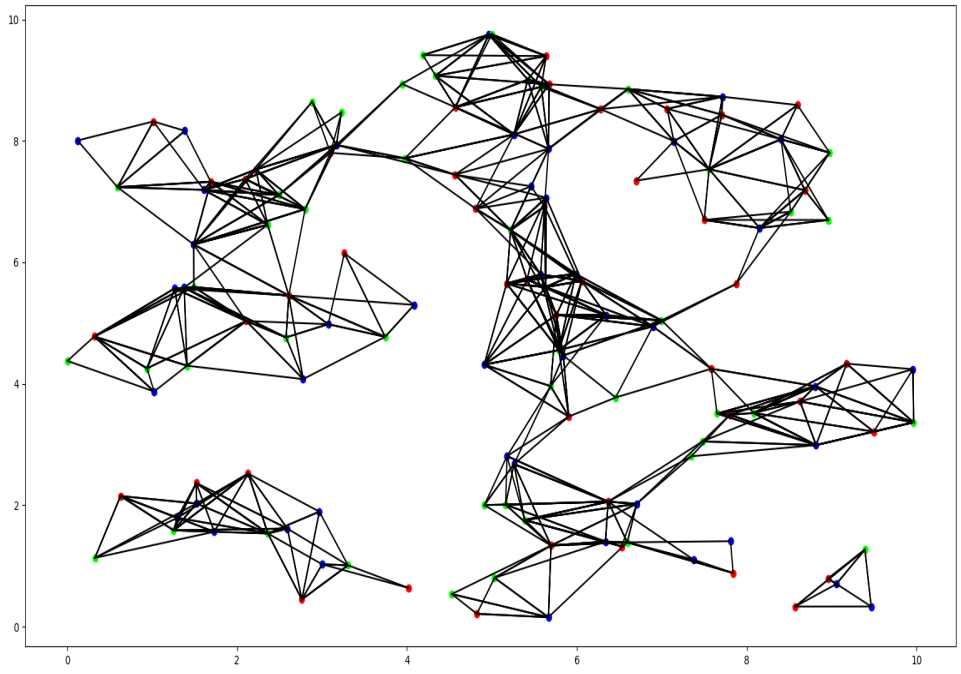


Figure 3: Triangle similarity join output.

8.3 First experiments

Using the same tests set used in previous section and a clique-4 shape, we report all the results in Table VIII, where RS is real shapes and AS is approximated shapes. Given the incredibly high number of solutions found by the algorithm on a dataset of 4000 points, we decided to skip the tests on such dataset.

TABLE VIII: TEST RESULTS FOR CLIQUE-4 SIMILARITY JOIN

R	#	ϵ	GCT[s]	QT[s]	RS	AS	AR
400 points							
1.5	2	0.1	0.087	0.105	8358	72	1.00861
1.0	2	0.1	0.04	0.032	711	10	1.01406
0.5	2	0.1	0.03	0.0	4	0	1.0
2.0	2	0.1	0.299	0.452	44471	126	1.00283
4.0	2	0.1	8.694	17.333	1977686	5396	1.00273
1.5	2	0.1	0.094	0.119	8358	72	1.00861
1.5	2	0.35	0.113	0.162	8358	2356	1.28189
1.5	2	0.065	0.093	0.128	8358	6	1.00072
1.5	2	1.0	0.392	0.783	8358	67242	9.04523
1.5	2	0.01	0.091	0.134	8358	0	1.0
1.5	7	0.1	0.022	0.0	0	0	1.0
1.5	3	0.1	0.031	0.003	41	0	1.0

1.5	4	0.1	0.02	0.0	0	0	1.0
1.5	2	0.1	0.099	0.138	9075	15	1.00165
200 points							
1.5	2	0.1	0.014	0.016	470	4	1.00851
1.0	2	0.1	0.01	0.003	47	0	1.0
0.5	2	0.1	0.008	0.0	1	0	1.0
2.0	2	0.1	0.033	0.044	2479	11	1.00444
4.0	2	0.1	0.519	0.969	109987	0	1.0
1.5	2	0.1	0.015	0.015	470	4	1.00851
1.5	2	0.35	0.017	0.017	470	60	1.12766
1.5	2	0.065	0.014	0.015	470	0	1.0
1.5	2	1.0	0.038	0.053	470	1829	4.89149
1.5	2	0.01	0.014	0.014	470	0	1.0
1.5	7	0.1	0.006	0.0	0	0	1.0
1.5	3	0.1	0.008	0.0	2	0	1.0
1.5	4	0.1	0.006	0.0	0	0	1.0
1.5	2	0.1	0.017	0.014	490	0	1.0

8.4 Second experiments

After the initial set of experiments, all the code has been updated to introduce another set of experiments more focused on the dynamic behavior of the algorithm, specifically a triangle

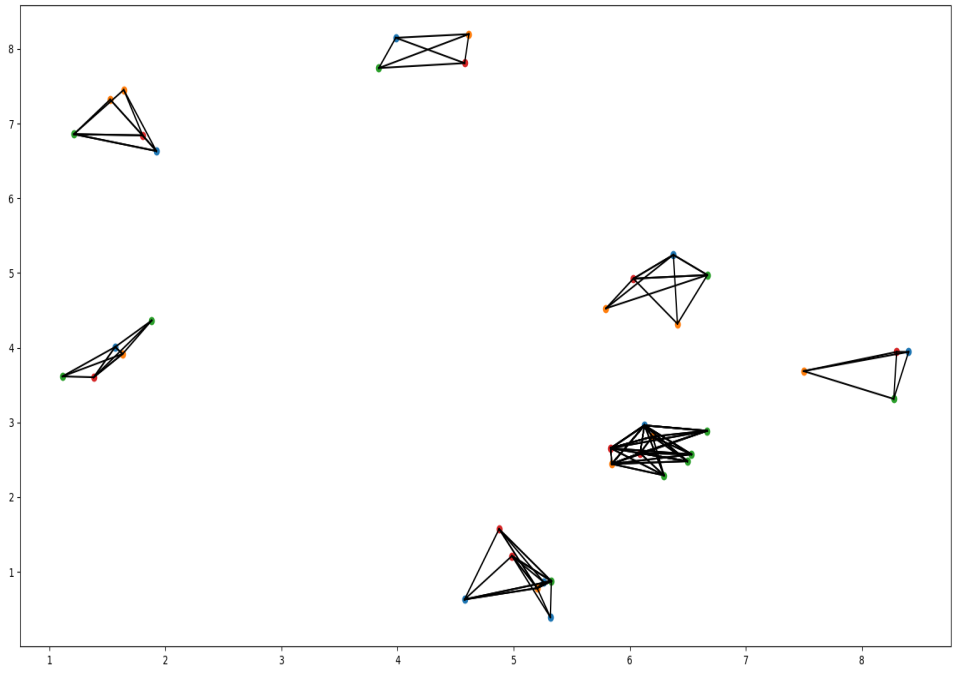


Figure 4: Clique-4 similarity join output.

similarity join has been run. Indeed a synthetic dataset, generated uniformly at random, and a real dataset have been used. In particular for the first one a dataset of 3000 2-dimensional points with coordinates from 0 to 10 have been used, while for the second one El-Nino tornado dataset[1] has been used.

The test set consists of the grid initialization with $init\_p\%$ of the original number of points; measurements on the update time and the query time follow. $step\%$ of the remaining points are added each step to the original dataset until all points are added. Each time we collect data about the query time and the update time, which is defined as the time to insert a single point into the current dataset.

8.5 Uniform dataset

As we can see from Figure 5, time needed to insert a point is increasing due to the fact that each time the algorithm has to work with an increasing number of cells. If all the cells available in the grid had been created, the update time would have been constant.

From Figure 6 and Figure 7 we can see how the query time is related to the output size, the more triangles are found the longer it takes to answer a query. Moreover in Figure 6 we can see some fluctuations given by internal factors of the system on which all the tests have been run. In Figure 7 the number of approximated triangles is always 0, because ϵ was set to a very small number.

---

[1] https://archive.ics.uci.edu/ml/datasets/El+Nino

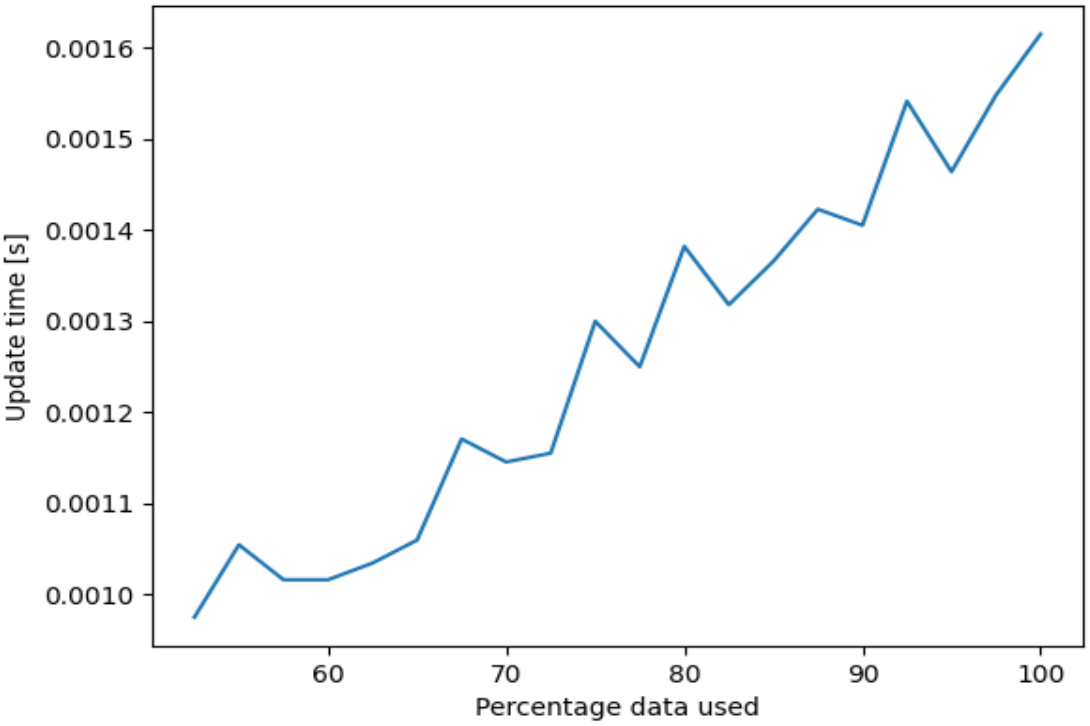


Figure 5: Update time graph for uniform dataset.

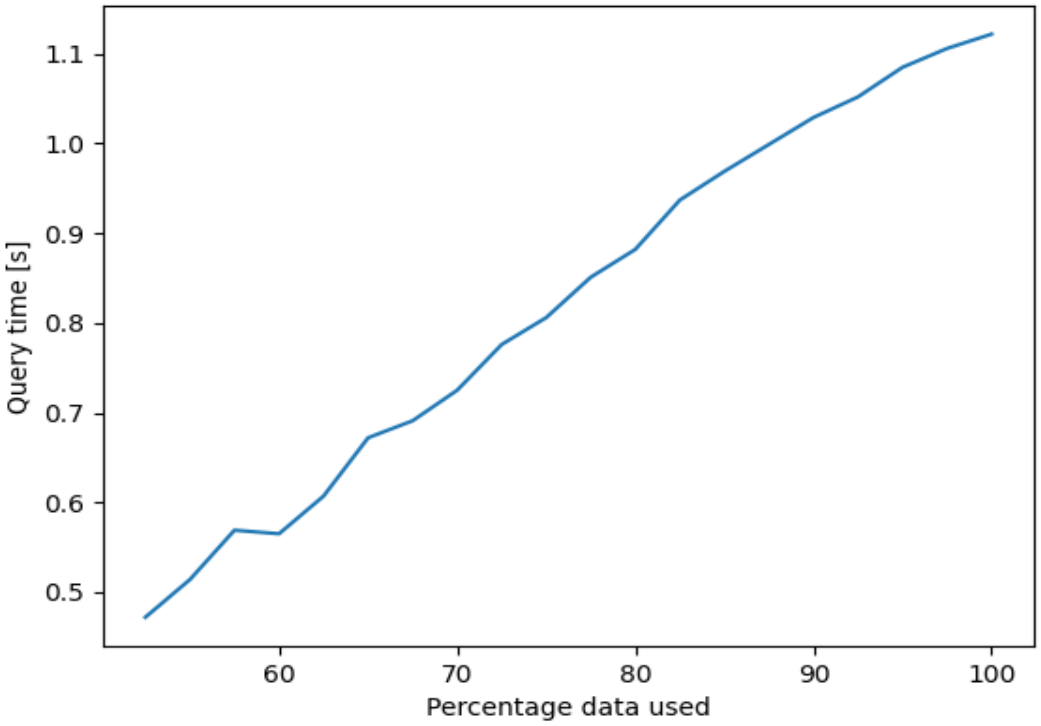


Figure 6: Query time graph for uniform dataset.

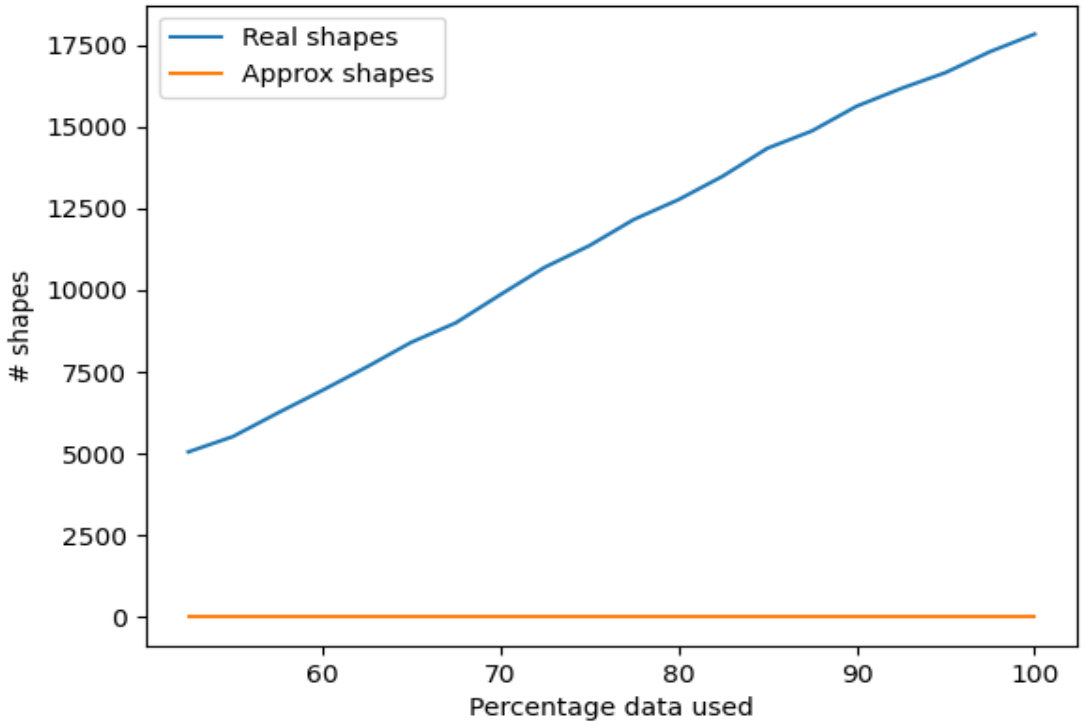


Figure 7: Real and approximated triangles graph for uniform dataset.

8.6 El-Nino dataset

The dataset contains the following attributes:

- year
- month
- day
- date
- latitude
- longitude
- zon.winds : zonal winds (west<0, east>0)
- mer.winds : meridional winds (south<0, north>0)
- humidity
- air temp.
- s.s.temp. : sea surface temperature

In the preprocessing phase all the years have been divided into three different intervals, which have been mapped to three different relations. Latitude and longitude of each record have been later inserted in the corresponding interval removing duplicates. Given that for attributes "year", "latitude" and "longitude" there are no missing values, the database for our algorithm is now ready to be processed, in Figure 8 an image of the processed dataset can be seen. If missing values are found, using different combinations of other attributes, we suggest to get ride

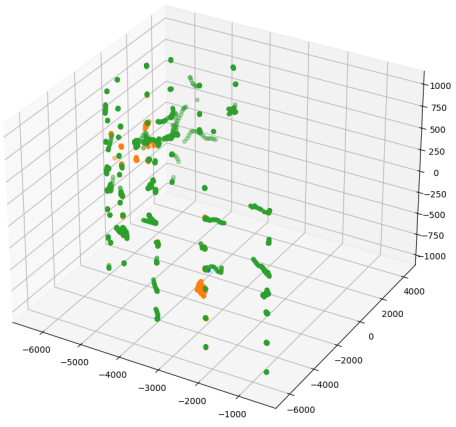


Figure 8: El-Nino Earth points by year intervals

of them with the minimum value of the same attribute among all the tuples. Finally a new boolean variable *GEO* has been added to Database.hpp file in order to switch from euclidean distance to the distance formula reported in the introduction to calculate the distance on Earth between two points described by latitude and longitude. Another script, plot_points_geo.py, has been added in order to plot in 3D space the points.

ϵ in this case is an approximation on latitude and longitude, which means that the approximation on the actual distance, using the previously described formula, will be at most:

$$\frac{\pi}{180}\epsilon * 6371$$

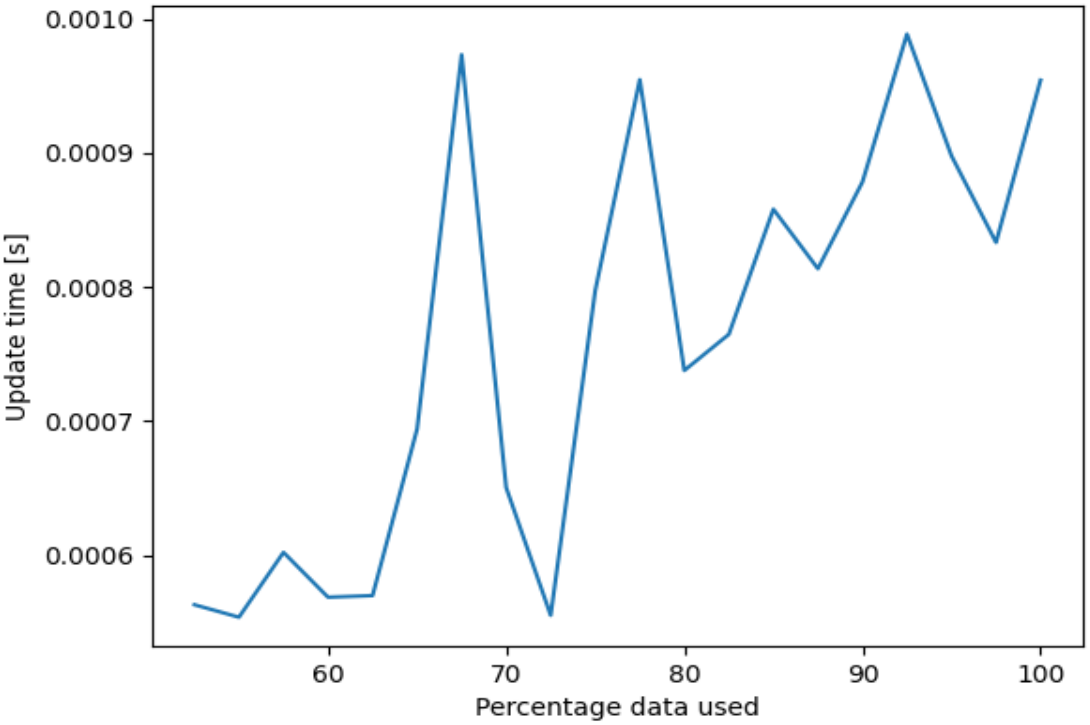


Figure 9: Update time graph for El-Nino dataset.

for both latitude and longitude, given that they are expressed in degrees. Indeed the worst case is represented by all cells lying on the equator, which are the biggest ones among all the others.

Running a similarity join on such database allows us to find locations close each other on Earth that have been exposed to tornadoes multiple times, so that we are able to find the most in danger areas.

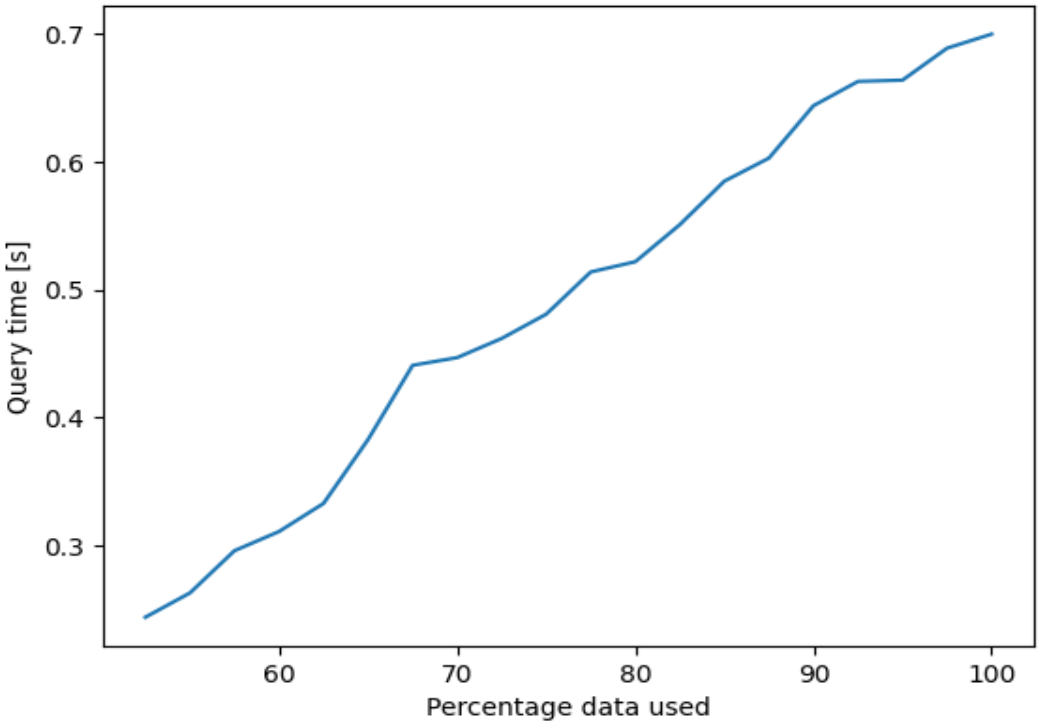


Figure 10: Query time graph for El-Nino dataset.

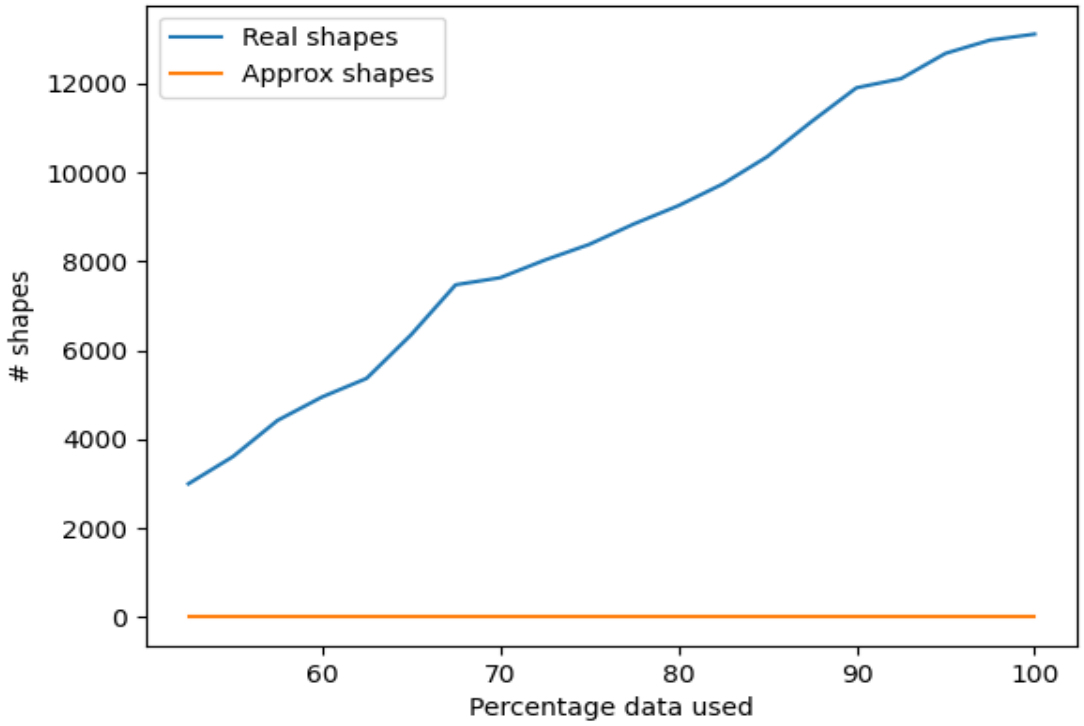


Figure 11: Real and approximated triangles graph for El-Nino dataset.

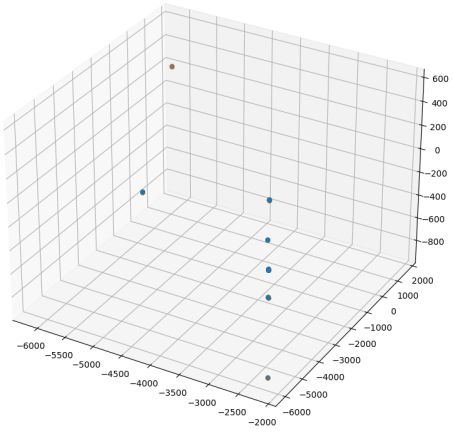


Figure 12: Locations on Earth with high tornado risk.

As in the uniform dataset case, we can see in Figure 9 how the update time is increasing. It means that after each update the algorithm has to work on a different number of cells that have been created in the previous update.

From Figure 10 and Figure 11 we can see, as in the previous case, that the query time depends on the output size, so the number of triangles that have to be reported, and that sometimes internal factors influence the algorithm running time.

Finally in Figure 12 we can see the locations on Earth that our algorithm found and whose population is very likely to be in danger for a tornado.

8.7 ϵ meaning in different metrics

As it has been highlighted in previous section, the approximation on the distance between each point is not always ϵ, but it depends on the distance metric used. Let's assume that the input space is represented by 2D euclidean space, that the output space is 2D euclidean space as well and that the distance metric used it l_2. Assuming each grid cell has diameter ϵ, without loss of generality, measuring the distance between (0, 0) and (ϵ,0) or (0,ϵ) leads to:

$$\sqrt{(0-\epsilon)^2 + (0-0)^2} = \epsilon$$

In general, given an input space $\mathcal{X}$ where the grid is built, an output space $\mathcal{Y}$ where distances are measured, ϵ an approximation constant in $\mathcal{X}$ and a distance metric $\mathcal{F}(a, b)$ between d-dimensional points a and b, the approximation δ on distance between points a and $a + \epsilon$ for each dimension in $\mathcal{Y}$ is given by:

$$\mathcal{F}((a_1, \ldots, a_d), (a_1 + \epsilon, a_2, \ldots, a_d)) : \text{for } d_1$$

$$\mathcal{F}((a_1, \ldots, a_d), (a_1, a_2 + \epsilon, \ldots, a_d)) : \text{for } d_2$$

$$\mathcal{F}((a_1, \ldots, a_d), (a_1, \ldots, a_d + \epsilon)) : \text{for } d_d$$

In order to better understand let's make an example related to the previously used dataset. On El-Nino dataset a grid with ϵ approximation has been built on 2-dimensional points represented by latitude and longitude. Such grid in $\mathcal{X}$ space, leads to cells with different shapes and dimensions in $\mathcal{Y}$. In Figure 13[1], we can see the cells mapped to $\mathcal{Y}$.

---

[1] https://commons.wikimedia.org/wiki/File:Division_of_the_Earth_into_Gauss-Krueger_zones_-_Globe.svg

Figure 13: Grid generated on Earth for El-Nino dataset.

CHAPTER 9

CONCLUSION

In this project we study the family of similarity join queries under any number of constant relations. In articular we show a conditional lower bound for the line–3 join in 3 dimensions, some conditions where line–3 join can indeed be solved exactly, a grid based data structure that works for any join graph and supports ε-approximate enumeration, and a reduction from similarity join to a set of equi-join queries. Finally, we implemented the grid-based data structure and we run experiments on synthetic and real datasets checking the efficiency and efficacy of our methods.

There are a lot of interesting open problems we are planning to work. It would be interesting to have a lower bound even when we cannot compute cos functions exactly. Geometrically, we are planning to study how close to the surface of the unit ball we can place points so that our reduction holds. Currently, our reduction consider the ℓ_2 distance. Another open problem is if the exact enumeration problem is expensive to solve for the ℓ_∞ distance. In particular, we would like to have a reduction where instead of balls we use squares to perform a reduction using the OuMv conjecture. Interestingly, it is known how to use the OuMv conjecture to show that the exact enumeration problem under intersection joins (general rectangles) is expensive to solve, however nothing is known about similarity joins under the ℓ_∞ norms (squares). Another interesting problem is to study the exact enumeration problem when the distance threshold r is

not known upfront, instead it is part of the query. Ideally, we would like to construct a dynamic data structure that can support ε-approximate r-enumeration for any arbitrary parameter r.

APPENDICES

CITED LITERATURE

1. Silva, Y. N., Aref, W. G., and Ali, M. H.: The similarity join database operator. In 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), pages 892–903. IEEE, 2010.

2. Henzinger, M.: Finding near-duplicate web pages: a large-scale evaluation of algorithms. In Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, pages 284–291, 2006.

3. Bayardo, R. J., Ma, Y., and Srikant, R.: Scaling up all pairs similarity search. In Proceedings of the 16th international conference on World Wide Web, pages 131–140, 2007.

4. Chaudhuri, S., Ganti, V., and Kaushik, R.: A primitive operator for similarity joins in data cleaning. In 22nd International Conference on Data Engineering (ICDE'06), pages 5–5. IEEE, 2006.

5. Chaudhuri, S., Ganti, V., and Kaushik, R.: Data debugger: An operator-centric approach for data quality solutions. IEEE Data Eng. Bull., 29(2):60–66, 2006.

6. Aiger, D., Kaplan, H., and Sharir, M.: Reporting neighbors in high-dimensional euclidean space. SIAM journal of computing, 43:pp. 1239–1511, 2014. A preliminary version of the paper has appeared in Proc. 24th ACM-SIAM Sympos. Discrete Algorithm, 2013.

7. Augsten, N. and Böhlen, M.: Similarity joins in relational database systems. Synthesis Lectures on Data Management, 5:1–124, 11 2013.

8. Chaudhuri, S., Ganti, V., and Kaushik, R.: A primitive operator for similarity joins in data cleaning. In 22nd International Conference on Data Engineering (ICDE'06), pages 5–5, 2006.

9. Agarwal, P. K., Hu, X., Sintos, S., and Yang, J.: Dynamic enumeration of similarity joins. CoRR, abs/2105.01818, 2021.

CITED LITERATURE (continued)

10. Khamis, M. A., Chichirim, G., Kormpa, A., and Olteanu, D.: The complexity of boolean conjunctive queries with intersection joins, 2021.

11. Berkholz, C., Keppeler, J., and Schweikardt, N.: Answering conjunctive queries under updates, 2017.

12. Tao, Y. and Yi, K.: Intersection joins under updates. Journal of Computer and System Sciences, 124:41–64, 2022.

VITA

NAME	Simone Zanella
EDUCATION	
	Master of Science in "Computer Science, University of Illinois at Chicago, May 2023, USA
	Specialization Degree in "Software Engineering", Jul 2023, Polytechnic of Turin, Italy
	Bachelor's Degree in Computer Engineering, Jul 2021, Polytechnic of Turin, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2021 - IELTS examination (7.0/9)
	A.Y. 2022/23 One Year of study abroad in Chicago, Illinois
	A.Y. 2021/22. Lessons and exams attended exclusively in English
	Fall 2020. Lessons and exams attended exclusively in English in Alabama
SCHOLARSHIPS	
Spring 2023	Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend
Fall 2022	Italian scholarship for TOP-UIC students
2018, 2019, 2021, 2022	Vitale Barberis Canonico's scholarship
2018	Confindustria's scholarship
TECHNICAL SKILLS	
Languages	C, C++, C#, Python, SQL, Rust, JavaScript, PHP, Java, HTML, CSS, Assembly
Frameworks	React.js, Express, Node.js

VITA (continued)

Libraries	React Router, Passport, PyTorch, scikit-learn, tslearn
Databases	MongoDB, Oracle, SQLite
Dev Tools	Visual Studio Code, Git, Gitlab, Android Studio, MATLAB, Unity

WORK EXPERIENCE

Jan 2023 - Present	Research Assistant
	Designed and developed a Machine Learning model to identify students with emotional issues based on their grades. Worked with unsupervised and semi-supervised learning for time series clustering. Built a user-friendly web interface to run the developed model.
Oct 2021 - Apr 2022	Research Assistant
	Worked with Moodle to provide a better learning experience to the end user. Developed a script for SCORM learning objects pre-processing in order to extract meta-data
Jun 2016 - Sep 2016 / Jun 2017 – Sep 2017	Web Developer
	Designed and developed a web page to import and process Excel files and insert data into an Oracle database. Worked with REST APIs to retrieve and display data from databases.

PROJECTS

OS161 Virtual Memory Manager	Designed and developed a Virtual Memory Manager for OS161 operating system. Implemented demand paging principles, TLB management, swapping, page replacement strategy and inverted page table.
React Study Plan Website	A CRUD application exposed using a RESTful API. Exposed POST, GET, PUT and DELETE HTTP methods using Express.
DodgeFireBall	Developed a game using Android Studio. Implemented threading for game management.
Conjunctive Query project	Implemented different algorithms for conjunctive query processing. Made some benchmarking on algorithms' execution time.

VITA (continued)

EzWh	Designed and developed a warehouse management system using Express and REST APIs. Used waterfall software development process. Applied project management techniques for team working. Drawn different types of diagrams like: class, context, deployment, design, use case diagrams. Acquired requirements conducting on-site interviews with fake costumers. Developed tests for verification and validation procedures.
Game Development	Designed and developed two games using Unity platform. Explored game dynamics.