Marius Aleksander Kaasbøll
Erik Nystø Rahka

Exploring Exemplar Trajectory Queries

Master's thesis in Computer Science
Supervisor: Kjetil Nørvåg
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Marius Aleksander Kaasbøll
Erik Nystø Rahka

Exploring Exemplar Trajectory Queries

Master's thesis in Computer Science
Supervisor: Kjetil Nørvåg
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Kunnskap for en bedre verden

Abstract

The proliferation of mobile devices enables people to log their geographical positions and to trace historical movements, which has spawned various novel applications. These applications have revealed new problems that require new solutions. One in particular is that of efficiently finding the top-k trajectories according to spatial- and textual similarity. An *exemplar trajectory query* is a query which specifies an ordered list of points in space, wherein each point has a textual description. The goal is to find the top-k trajectories as ranked by a given similarity function. In this thesis we explore and attempt to implement two algorithms used two answer such queries. We will also attempt to extend one of these algorithms to process large volumes of data using Apache Spark.

Sammendrag

Utbredelsen av GPS-kompatible mobile enheter som lar brukere loggføre sine geografiske posisjon samt spore bevegelser over tid, har gitt opphav til flere nye ulike applikasjoner. Disse applikasjonene har belyst nye problemer og utfordringer som krever nye smarte løsninger. Et eksempel på et slikt problem, er hvordan vi effektivt kan finne de k-likeste *trajectories* til en gitt *trajectory*. Et *exemplar trajectory query* er enn slik type spørring. Denne spørringen består av en samling med punkter i en gitt rekkefølge, hvor hvert punkt har en tekstlig beskrivelse. Resultatet av en slik spørring vil være de k likeste *trajectories* gitt ved en similaritetsfunksjon. I denne rapporten vil vi undersøke og implementere to algoritmer som kan brukes for å svare på slike spørringer. Videre vil vi utvide en av algoritmene til å håndtere store datamengder ved hjelp av Apache Spark.

Contents

Abstract . . . iii
Sammendrag . . . iv
Contents . . . v
1 Introduction . . . 1
1.1 Research Questions . . . 2
2 Preliminaries . . . 3
2.1 Definitions . . . 3
2.2 Problem definition . . . 5
3 Background . . . 6
3.1 Querying spatio-textual data . . . 6
3.1.1 Processing Top-k spatial keyword queries . . . 6
3.1.2 RCA: Rank-aware Combined Algorithm . . . 7
3.1.3 k-BCT Queries . . . 7
3.2 Distributed Computing . . . 8
3.2.1 Apache Hadoop . . . 8
3.2.2 Apache Spark . . . 9
4 Exemplar Trajectory Query . . . 11
4.1 ILA . . . 11
4.1.1 Upper boundary of similarity for unseen trajectories . . . 14
4.1.2 Lower bound of similarity for seen trajectories . . . 15
4.1.3 Upper bound of similarity for seen trajectories . . . 15
4.1.4 Maximum number of iterations . . . 16
4.1.5 Optimizations to ILA . . . 17
4.2 2TA . . . 19
4.2.1 Posting list . . . 20
4.2.2 Grid index . . . 21
4.2.3 Processing ETQ with 2TA . . . 21
4.2.4 Upper boundary of unseen trajectories . . . 23
4.2.5 Upper boundary of textual similarity . . . 23
4.2.6 Upper boundary of spatial similarity . . . 24
4.2.7 Upper boundary of similarity for any unprocessed points for q_i . . . 24
4.2.8 Upper boundary of similarity for seen trajectories . . . 24
4.2.9 Optimizations to 2TA . . . 26
5 Parallel ETQ processing . . . 28
5.1 2TAP: Multithreaded 2TA . . . 28
5.2 Ellsworth: ETQ on Spark . . . 29
5.2.1 Partitioning scheme . . . 29
5.2.2 Query execution . . . 30
5.3 Naive ETQs on Spark . . . 32

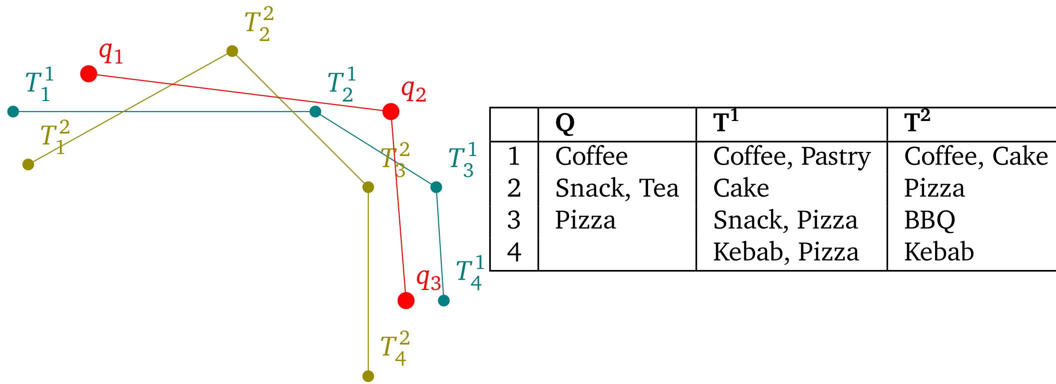
6 Experiments **33**
6.1 Setup 33
6.1.1 Environments 33
6.1.2 Datasets 34
6.2 Comparing ETQs 35
6.3 Scaling ETQ 38
6.4 Distributed ETQ 41
7 Discussion **44**
7.1 Re-implementing ILA and 2TA 44
7.2 Parallelizing 2TA 45
7.3 Developing Ellsworth 45
8 Conclusion **47**
8.1 Further work 47
Bibliography **49**
9 Appendix **51**
A Edge cases in ILA and 2TA **52**

1. Introduction

Large amounts of geographically-tagged data is being recorded every second all over the globe. Whenever we use a GPS-enabled device, such as a smart-phone, we leave behind a trail of geographical data points. Twitter for instance, creates large collections of *spatio-textual* data, as tweets contain text and not uncommonly- a geotag. A study in 2013 found that 0.85% of tweets are explicitly geotagged, which may seem like a small amount [1]. However, Sloan and Morgan estimated that in 2016 over 500 million tweets were produced per day, meaning 0.85% still amounts in over 4 million geo-tagged tweets every 24 hours [2]. Other social media, such as Facebook or Instagram also produce similar data. If we string a series of these *spatio-textual points* together, we get a *spatio-textual* trajectory. This sheer volume of spatio-textual trajectories accumulates to large amounts of data, which introduces problems that requires new and efficient solutions.

There are several different types of queries that one can perform on such data. One example is the *top-k spatial keyword* query, which aims to locate the top-*k* points based on a similarity measure which considers both textual- and spatial components [3]. Yanagisawa *et al.* presents another form of spatio-textual query which locates a trajectory that best matches the *shape* of an input query. Yet another form of spatio-textual query is the *exemplar trajectory query* [5]. An exemplar trajectory query (ETQ) can be considered an extension to the *top-k spatial keyword* query, but which operates on trajectory data, rather than standalone points. Wang *et al.* presents a solution to this type of query in "Answering Top-k Exemplar Trajectory Queries" [5]. In this thesis, we will focus on the *exemplar trajectory query*.

An example use-case for exemplar trajectory queries is trip-planning. Consider the following; you're on a trip to a foreign city, and you wish to visit a series of places, and perhaps find something to eat or do around these points. You have a trip-planner application, which is linked to a database containing check-in's or routes through the city, as recorded by other tourists or locals. You could then query this database using an *exemplar trajectory query*, where you would input the points you wish to visit, along with the activities you wish to do near said points. The ETQ processor would then find the existing paths (trajectories) in its database which best match your query, both considering the points' position and their activities. An example is shown in Figure 1.1.



	Q	T¹	T²
1	Coffee	Coffee, Pastry	Coffee, Cake
2	Snack, Tea	Cake	Pizza
3	Pizza	Snack, Pizza	BBQ
4		Kebab, Pizza	Kebab

Figure 1.1: Spatio-textual trajectories T^1, T^2 and an exemplar query trajectory Q. Their respective terms are shown in a table on the right.

1.1 Research Questions

In this thesis, we investigate how the methods for solving *exemplar trajectory queries* (ETQ) by Wang *et al.* can be adapted for operating in a distributed environment to provide better scaling when working with large volumes of data. In order to accomplish this we implement two of these algorithms, adapt one of them to run on a cluster, and evaluate these against one another using a collection of datasets of varying sizes.
Throughout this thesis, we attempt to answer the following research questions:

RQ1. Are the algorithms for answering *exemplar trajectory queries* by Wang *et al.* reproducible?

RQ2. How do these algorithms scale with larger volumes of data?

RQ3. Can *exemplar trajectory queries* perform better at scale by utilizing distributed computing?

The first part of this thesis explores the existing work within the field of spatio-textual trajectory processing, and defines what a spatio-textual trajectory is. We also look into systems for performing distributed computations. In chapter 4 we investigate the algorithms presented by Wang *et al.*; the *Incremental Lookup Algorithm* and the *Two-level Threshold Algorithm*, as well as some of the considerations made while implementing these ourselves. Chapter 5 details how we created *Ellsworth*, an adaption of the *Two-level Threshold Algorithm* designed to run in a distributed cluster. Chapter 6 compares the performance of our implementations of the *Incremental Lookup Algorithm* and the *Two-level Threshold Algorithm* to the original results. We also perform scalability tests and evaluate the performance of *Ellsworth*. Chapter 7 describes our experiences, challenges and further analyzes the results we found. Finally, in chapter 8 we summarize our findings.

2. Preliminaries

This chapter will briefly explain some preliminary information, such as definitions of similarity, and a formal problem description. The chapter also includes a table of common notations used throughout the thesis.

2.1 Definitions

We define a spatio-textual point, referred to as a point. A point p is made of two components. A spatial component and a textual component. The spatial component $p.loc$ is a geographic position, represented by a latitude and longitude pair. The textual component is a set of terms $p.terms$. $p.terms$ is a *set*; meaning it is a collection of distinct terms $(t_1, t_2, \ldots, t_n)$.

We also define a *spatio-textual trajectory*, referred to as a trajectory. A trajectory T is an ordered set of n spatio-textual points $(p_1, p_2, \ldots, p_n)$. A query Q has the same definition as a trajectory, but is denoted as Q. A query Q is a ordered set of m spatio-textual points $(q_1, q_2, \ldots, q_m))$.

In order to be able to compare trajectories, we need a couple of definitions to formalize what a similarity between two trajectories actually represent. Note that there are various similarity-measures that can be used to compare trajectories, but since our work is based on the work of Wang *et al.* [5] we've elected to use a similar set of similarity definitions.

Definition 1 (Spatial similarity). *The Euclidean distance is used to measure spatial similarity:*

$$\hat{S}_S(p_i, p_j) = \frac{D_{\max} - Euclidean(p_i, p_j)}{D_{\max}}$$

The variable $D_{\max}$ is the maximum distance between any two points in the dataset and is used to normalize the spatial similarity between 0 and 1 [5, Definition 4].

Note that whilst Euclidean distance is used in the original work, one can use other distance metrics. The euclidean distance is not as accurate as a metric compared to great-circle distance, particularly at high latitudes or when applied to trajectories that cover longer stretches around the globe. Regardless of which distance-function is used, it is important that the $D_{\max}$ variable is calculated the same way. In our work we chose to use euclidean distance.

Definition 2 (Textual similarity)**.** *Textual similarity is defined as the sum of textual relevance of each term t that is common between p_i and p_j [5, Definition 4].*

$$\hat{S}_T(p_i, p_j) = \sum_{t \in p_i.terms \cap p_j.terms} tf(p_j, t) \cdot idf(t)$$

Here $tf(p_j, t)$ is the raw frequency of term t in $p_j.terms$ and $idf(t) = \log(\frac{N}{n_t})$, where N is the total number of points in $D.P$ and n_t it the number of points that contains the term t [6].

Wang *et al.* uses a simple TF·IDF model to calculate the weight of term t in p_j, although the exact TF·IDF model is not specified. We chose to use raw frequency as term frequency, and inverse frequency as inverse document frequency [6]. The TF·IDF weight of term t in point p is denoted as $\gamma(p, t)$.

We also normalize the TF·IDF weights between 0 and 1 by using the respective point's euclidean norm (L_2 norm) [7]. The normalization is done by first calculating a *norm* value for point p, which is found by summing the squares of each TF·IDF weights in the given point. The normalized score for term t in p is calculated by dividing the TF·IDF weight of t in p by the *norm* value.

$$\gamma_{norm}(p, t) = \frac{\gamma(p, t)}{\sqrt{\sum_{i=1}^{|p.terms|} \gamma(p, p.terms[i])^2}}$$

Note that because a point p contains a **set** of terms, any term t cannot occur more than once in $p.terms$. This causes the term frequency to always be equal to 1 ($\forall p \in D.P \ \forall t \in p.terms \ tf(p, t) = 1$).

Definition 3 (Point-to-point similarity)**.** *The similarity between two points p_i and p_j is defined as [5, Definition 4]:*

$$\hat{S}(p_i, p_j) = \begin{cases} 0, p_i.terms \cap p_j.terms = \emptyset \\ \alpha \cdot \hat{S}_S + (1 - \alpha) \cdot \hat{S}_T, otherwise \end{cases}$$

Definition 4 (Point-to-trajectory similarity)**.** *The similarity between a query point q_i and a trajectory T is defined as [5, Definition 5]:*

$$\hat{S}(q_i, T) = \max_{p_j \in T} \left\{ \hat{S}(q_i, p_j) \right\}$$

Definition 5 (Point-wise similarity)**.** *Point-wise similarity between T and Q is defined as the sum of the point-to-trajectory similarities between T and each point in Q, normalized by $|Q|$ [5, Definition 6]:*

$$\hat{S}(Q, T) = \sum_{q_i \in Q} \hat{S}(q_i, T) / |Q|$$

The variable $\alpha = \langle 0, 1 \rangle$ is a tuning parameter that decides the weight of the textual similarity versus spatial similarity of a point. A higher α value emphasises the spatial aspect. When $\alpha = 1$ the similarity will ignore the textual aspect. The inverse applies to textual similarity, when $\alpha = 0$ the similarity will ignore the spatial aspect.

Notation	Explaination
p	A point, consisting of a lat-long location *loc*, and a set of *terms*
$p.terms$	The set of terms $(t_1, t_2, ..., t_n)$ belonging to a point p.
T	Trajectory, collection of points $(p_1, p_2, ..., p_n)$
D	Dataset, a collection of trajectories
$D.P$	Refers to all points within a dataset
Q	An exemplar trajectory query
$\hat{S}_S(q_i, p_i)$	Spatial similarity between point q_i and p_i (Definition 1)
$\hat{S}_T(q_i, p_i)$	Texual similarity between point q_i and p_i (Definition 2)
$\gamma(p, t)$	TF·IDF weight of term t in point p
$\hat{S}(Q, T)$	Point-wise similarity between trajectory Q and trajectory T (Definition 5)
D_{max}	Maximum distance between any pair of points within a dataset
α	Tuning parameter for textual- versus spatial similarity.

Table 2.1: A summary of notations used throughout this thesis

Lastly, we define form of spatio-textual query called a *top-k spatial keyword query*, referred to as a TkSK. This is a query that retrieves points based on both their textual- and spatial relevance to a query point. Zhang *et al.* defines a TkSK as [8]:

Definition 6 (Top-k Spatial Keyword Search)**.** *Given a document corpus $D.P$, a top-k spatial keyword query Q retrieves a set $O \subseteq D.P$ with k documents such that $\forall \mathcal{D} \in O$ and $\mathcal{D}' \in D.P - O$, $\hat{S}(\mathcal{D}, Q) \geq \hat{S}(\mathcal{D}', Q)$ [8, defenition 1].*

2.2 Problem definition

Finally, we define the exemplar trajectory query. An exemplar query trajectory Q is like any other trajectory, a collection of points $(q_1, q_2, \ldots, q_n)$. When answering a *top-k exemplar trajectory query*, we aim to find the k most similar trajectories to a query-trajectory Q in D according to a given *point-wise* similarity function $\hat{S}(Q, T)$ in a trajectory database D.

3. Background

The following chapter consists of two sections where we will investigate existing work within the field of spatio-textual data processing. First we'll look at some solutions for querying spatio-textual data; both point data and trajectories. Next we'll examine frameworks for performing computations at a larger scale, by distributing the load of querying across multiple nodes of a cluster.

3.1 Querying spatio-textual data

There are several different types of queries that are potentially interesting to perform on a database of spatio-textual data. One might wish to query based on the shape of data, the locality of data, or even textual matches. There are also several query-types which consider the dual-nature of spatio-textual data; for instance a top-k spatial keyword query, or an exemplar trajectory query. This section will elaborate on some methods used query spatio-textual data collections.

3.1.1 Processing Top-k spatial keyword queries

One form of spatio-textual query is a *top-k spatial keyword query*, referred to as a TkSK. Rocha-Junior *et al.* explores solutions to query spatio-textual data points based on a similarity function that considers both the textual and spatial aspects of a point [3].

Rocha-Junior *et al.* introduces a novel index named *spatial inverted index (S2I)* alongside two algorithms, *SKA* and *MKA*. A key trait of the S2I index is how it considers the frequency of a term when storing it. Zipf's law states that there is a small number of terms which occur very often, while most terms are infrequent [9]. S2I exploits this by having different datastructures, depending on the frequency of a term. At its root, S2I is a form of inverted index, where the key is a given term. However, depending on the frequency of the term, its value can either be an unordered list of points (block), or a pointer to an aR tree. A term which occurs infrequently is stored in simple blocks, which are unordered list. A term with a higher number of occurrences on the other hand, maintains an *aggregated R* (aR) tree which acts as a subindex for all the points which reference the term. An aR tree is a specialized form of the well known R-Tree, described by Papadias *et al.* [10]. It distinguishes itself form a traditional R-Tree by allowing nodes to store a non-spatial value based on an aggregation of its sub-nodes [10]. In S2I, this value is used to store the maximum impact of a term on the point in a nodes sub-tree. This way, S2I can explore the tree spatially, while still being mindful of the textual property.

Next, they propose two algorithms for querying the S2I index: SKA, Single Keyword Algorithm; and MKA, Multiple Keyword Algorithm. SKA only needs to access a single term within the index, which translates to a single block or tree. Should the term point to a block of points, it maps each point to a heap based on a similarity function that compares it to the query point. After, it simply returns the top-k results. If the key maps to an aR tree on the other hand, it uses an incremental algorithm to iteratively return points in decreasing order, according to a given similarity-function. MKA processes query points differently, by splitting the processing into two parts. The first phase computes *partial-scores*: the score of a point according to a single query term. The second phase aggregates the partial-score results to retrieve the top-k matches.

Rocha-Junior *et al.* concludes that the MKA and SKA algorithms in conjunction with the S2I index can perform TkSK queries at a rate that outperforms existing state-of-the-art solutions. They are found to be efficient both in terms of query time, and the cost of updating the index upon insertion of data.

3.1.2 RCA: Rank-aware Combined Algorithm

The RCA algorithm proposed by Zhang *et al.*, is a rank-aware adaptation of the CA-algorithm [11]. This is another solution for *top-k spatial keyword* queries. Specifically, Zhang *et al.* examine the problem of retrieving a ranked set of entities, based on both textual similarity and spatial proximity [8].

A core principle of RCA is the notion of *score-bounded expansion*. The algorithm should explore its search-area in a manner which ensures elements with high similarity are discovered before low-similarity elements. It does this by maintaining two separate indexes; one where data is indexed by their spatial attributes, and another index that allows lookup based on textual attributes.

The spatial index used by Zhang *et al.* is a Z-order curve grid-index. A Z-order curve grid-index is a grid-index using the Morton order space filling curve to map n-dimensional data to a single dimension. The *Morton code* is easily encoded by interleaving the bits for each coordinate value [12]. The Z-order curve grid-index implementation should allow for incremental expansion of a search-area around a given point. With this, the algorithm can efficiently access points around the query point in an incremental manner, where nearby points are discovered sooner than faraway points. Textual attributes are indexed using a ranked inverted list for each term in the dataset, which they call *posting-list*. For every term t, the points where $t \in p.terms$ are sorted by TF·IDF weight in descending order. By iterating block-wise over these posting-lists, textually relevant points can be accessed early in the process.

3.1.3 *k*-BCT Queries

Chen *et al.* proposes a k-BCT query, or k best connected trajectories query. This query-type operates on entire trajectories, rather than single points. However it only considers the spatial aspect of the data. It is described as a form of trajectory query that attempts to find the k nearest trajectories to a set of query points [13]. Zheng and Zhou[14] defines the distance between a query location q_i and a trajectory $T = (p_1, p_2, \ldots, p_n)$ as follows:

$$Dist_q(q_i, T) = \min_{p_j \in T}\{Dist_{euclidean}(q_i, p_j)\} \qquad Dist_q(Q, T) = \sum_{i=1}^{|Q|} e^{-Dist_q(q_i, T)}$$

In other words, the trajectory-to-trajectory similarity is computed by first finding the nearest query point for each point in the trajectory and calculating the spatial similarity for said pair, then finally summing the best similarities for each query point.

Chen *et al.* proposed an incremental approach to finding top-k best connected trajectories for a given set of query points, an algorithm named *Incremental KNN-based Algorithm* or IKNN [13]. The algorithms runs on top of a R-Tree spatial index. When performing a query, it uses a series of k-nearest-neighbor queries to expand the search area around each point of the query trajectory. This is done by traversing and pruning the R-Tree index. By incrementing the k-value used for the KNN search, the algorithm incrementally increases the search space around each query point until a set of boundary-conditions are satisfied. At that point, one can be sure all the top-k trajectories have been located, and candidate set is run through a final sort before the top-k best connected trajectories are returned [13].

3.2 Distributed Computing

Distributed computing is a common way to provide *horizontal* scaling for both storage, and computationally intensive applications. By distributing the load across multiple machines, one can use off-the-shelf components to perform computation at a large scale without necessarily needing to resort to expensive hardware. However, distributing an application to run across multiple nodes presents several challenges, such as data- and task-partitioning, heterogenous nodes, and fault-tolerance[15]. In this section, we will present some frameworks which make distributed computations more accessible.

3.2.1 Apache Hadoop

'The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models'. [16]. The Hadoop framework includes a series of main modules: HDFS, a distributed file system; Yarn, a job scheduling- and cluster managing framework; and MapReduce, which is used to process large datasets in parallel. There are also a number of other projects related to the Hadoop project at Apache which support a larger variety of workflows and use-cases. One example is Apache Spark, further detailed in section 3.2.2.

The primary computing framework in the Apache Hadoop project is an implementation of *MapReduce*. MapReduce is a programming framework [17], which has since its inception been used and implemented by multiple parties, such as by Google and as a part of the Apache Hadoop platform [17]. The MapReduce programming model operates on a set of key-value data, and two central functions:

map$(k_1, v_1) \rightarrow (k_2, v_2)$: The *map* function is a user-defined transformation, changing a key-value pair into a new intermediate key-value pair. The MapReduce implementation then groups all intermediate pairs with matching keys together, and passes them to the *reduce* function.

reduce$(k_1, list(v...)) \rightarrow (k_1, v_3)$: The *reduce* function is another user-defined transformation, which receives a key, and set of values associated with said key. The *reduce* functions job is to perform a reduction: reducing the input values to a smaller set, or commonly a single value. The resulting value is then persisted to disk or returned to the calling application.

Apache's MapReduce implementation provides a task-scheduler, which assigns work to each worker-node in the cluster. The task-scheduler partitions data, manages liveness of each node, and manages fault-tolerance. By default, data is partitioned using a hash-partitioner, which uses a hash-value based of the `key` in the key-value pair to compute a partition key. This provides a fairly uniform distribution of the data and should lead to fairly uniform distribution of work. The task-scheduler also provides a slew of functionality to make distributed computing run smoothly. An example feature is its ability to run backup-tasks: in the cases where a node falls behind, referred to as a "straggler", the task-scheduler can issue a duplicate (backup) of the stragglers' task to another node. A "straggler" can occur for any number of reasons, such as a bad disk, or poor bandwidth to the particular node. The task-scheduler also considers data-locality when run on a supported filesystem, allowing the computation to be brought close to the data. An early implementation of a MapReduce framework, described in [17] supports GFS [18], but later implementations support a greater variety of distributed filesystems, such as the Apache Hadoop Filesystem (HDFS).

3.2.2 Apache Spark

Zaharia *et al.* [19] introduces a framework for performing cluster-computing on large sets of what they call *working datasets*. A *working dataset* refers to a dataset that is reused within the same application- such as for repeated transforms in an iterative task, or during interactive querying. Hadoop MapReduce struggles in these workflows, due to MapReduce persisting and loading data from disk in between tasks. Loading the same dataset over and over for an iterative task would cause high disk IO, and quickly becomes a major performance bottleneck. Zaharia *et al.* lists two primary motivations behind Spark: *interactive querying* and *iterative jobs*. Spark also aims to deliver the same scalability and fault tolerance as MapReduce, without incurring the same latency when querying a *working dataset*.

Several Spark abstractions, such as the DataFrame API[20], or Spark Streaming [21], run on top of a central Spark concept: *Resilient distributed datasets*, or RDDs. RDDs are described as a distributed memory abstraction that allows for cluster computations on large sets of data in a fault-tolerant manner. Zaharia *et al.* defines an RDD as a read-only, partitioned collection of records, which can only be created through deterministic operations on either (1) data in stable storage or (2)other RDDs [22]. An RDD is an immutable set of data, and can therefore reliably be used in a number of parallel operations without the risk of data-loss through write-conflicts. This property also gives RDDs a high-level of consistency compared to other distributed shared-memory systems.

An RDD is composed of multiple partitions, where each partition is an atomic part of a dataset. Each RDD also has a set of dependencies to one or more parent RDDs and a *compute()* function that is used to derive the RDD's data from the data of the parent RDDs. As RDDs are immutable structures, they are not altered directly, but rather transformed though a series of operations- such as *map*, *filter* or *reduce*. When a transformation is applied, a new RDD is created from the partitions of the previous RDD. For example, `map` creates a *MapPartitionsRDD* which carry over properties from its parent RDD, and applies a given `map` function to each entry in its data. While the partitions are invisible to many transformations such as the ones mentioned, transformations like the *mapPartitions* can discern that values belong to distinct partitions and performs a "partition local" map. Instead of transforming each

value one by one, it can operate on all values in a partition as a whole. Note that RDDs are computed lazily, so generally an RDD is only materialized when an *action* is called (*collect*, *count*, *take*) to retrieve results. If one needs to reuse the same working data for another query, data can be persisted in memory (and optionally spilled to disk) using the `.cache()` function. Each transformation is logged to provide a *lineage* for an RDD. Should a partition or even an entire RDD be lost, its *lineage* can be used to recompute the lost data. MapReduce on the other hand, needs to persist each intermediate step of transformed data to disk, requiring additional disk-storage and causes latency through disk IO.

Operations like *map* and *filter* will result in RDDs with a single parent dependency, while other operations such as *join* joins two RDDs into a single RDD which will be dependent on both of the participating RDDs. We can classify dependencies as either narrow dependencies or wide dependencies. In the case of a narrow dependencies, all partitions within a parent RDD is used by at most one single child partition, whilst wide dependencies indicate that parent partitions are used by multiple child partitions. Narrow dependencies result in tidy lineages that can easily be computed as the parent partitions is present on the same node. In wide dependency operations, child partitions require multiple parent partitions to be present, all of which may not be present on the same node, which may require data to be transferred amongst nodes. This is called a *shuffle*, and is significantly more expensive than the narrow dependency counterpart [22].

Well-thought-out partitioning of data can reduce shuffling further down the line [23]. RDDs commonly inherit their parent-RDDs partitioner, but one can also explicitly repartition the data as needed. When loading datasets from HDFS for instance, the data is partitioned based on the HDFS blocks, making it easy to bring the computations to the data, rather than needing to move the data across the network. Another partitioner is the HashPartitioner, which distributes data based on computed hash-values for the data. Users can also implement their own partitioner, allowing the user to apply their domain-knowledge to tailor the partitioning-scheme and distribute data in a manner optimized for a particular application. Some operations can leverage such a partitioner, if two RDDs share the same partitioner they are *co-partitioned*. Operations like *join* can take advantage of co-partitioned RDDs to perform a co-partitioned join, an optimized join that creates a narrow dependency between the resulting child RDD and the parent RDDs. This is possible as all values to be joined are guaranteed to reside in the same partition. All of the values in a single partition on one of the parent RDDs are guaranteed to be in a single partition in the other parent RDD. Such a join reduces the number of shuffles required [24].

RDDs also lay the foundation for parallelism within Spark. Each RDD partition represents a unit of work, called a *task*. Tasks that can be executed in parallel are referred to as a *stage*, and a *job* is a sequence of tasks. Whenever an *action* is called to materialize an RDD, the driving application starts a *job*, which causes each worker node to perform the specified transformation to their partitions, before finally collecting the result in the *driver* application. The driver application is also responsible for requesting cluster-resources, such as the number worker instances, or the heap size for each instance. Tuning these parameters are important for the applications' performance, as a single core can process one task at a time, i.e. one partition. This makes the number of partitions another important consideration- if the number of partitions is smaller than the number of executing cores, leftover cores will remain idle, and reduce the overall performance of the application.

4. Exemplar Trajectory Query

Given a database of trajectories and a query trajectory, an **exemplar trajectory query** (ETQ) finds the top-k most similar trajectories in a trajectory-database according to a point-wise spatio-textual similarity function, as defined in definition 5. An ETQ accepts a query trajectory Q and number k which specifies how many trajectories should be retrieved. The output is a list of the k best matching trajectories, in no particular order.

Wang *et al.* presents three algorithms for performing ETQs: the *Incremental Lookup Algorithm (ILA)*, the *GAP-bounded Incremental Lookup Algorithm (GAP-ILA)*, and the *Two-level Threshold Algorithm (2TA)*. For the purpose of this thesis we will focus solely on 2TA and ILA, which will be described in detail in this chapter. It is worth mentioning that they have demonstrated that both of these algorithms can be extended in order to perform order-sensitive queries, but this is not examined in this thesis. The algorithms, definitions, and methods described in this chapter are based of the work of Wang *et al.* [5]

4.1 ILA

The incremental lookup algorithm (referred to as ILA) expands upon the IKNN (Incremental nearest neighbor algorithm) proposed by Chen *et al.* in [13], taking it from a spatial-only algorithm to a spatio-textual algorithm. It uses a *top-k spatial keyword search (TkSK)* to search for and retrieve points that are similar to each query point $q_i \in Q$. We can then derive a candidate set of trajectories from the retrieved points by mapping the points to their parent trajectory. An important property of a TkSK is that all retrieved points must have a better similarity to their respective query point than any point that was not retrieved (all remaining points). This property of a TkSK enables us to calculate a set of boundaries of similarity between each seen trajectory and the query points. (1) Seen lower bound: The minimum possible similarity a seen trajectory can have. (2) An upper bound of similarity between each seen trajectory and the query, representing the maximum possible similarity a seen trajectory can have. (3) An upper bound of similarity for any unseen trajectory, which is the maximum possible similarity between the query and any trajectory not contained in the candidate set.

By using these boundaries we can limit the search space by incrementally expanding it until we know that the similarity of any **seen** trajectory cannot beat the similarity of any **unseen** trajectory; if at least k candidate trajectories has a lower bound of similarity higher than the upper bound of similarity for any unseen trajectory then we know that the top-k trajectories is guaranteed to be contained in candidate set. If there are less than k candidates with high enough lower bound then we must expand our search space in order fetch more points which leads to more candidates. This is repeated until at least k candidates have high enough lower bound. Afterwards the actual similarity between the query and every trajectory in the candidate set can be calculated and sorted to retrieve the top-k trajectories.

```
1   function ila(query, k)
2       result = []
3       λ = k
4       c = 0
5       λ^max = calculateMaxLambda() //Definition 11
6       R = []
7       λ_queried = []
8       while λ ≤ λ^max
9           R[c] = []
10          foreach q_i ∈ query
11              λ_i^max = calculateMaxLambda(q_i) //Definition 10
12              if λ ≤ λ_i^max or c = 0 or λ_queried[q_i] < λ_i^max
13                  R[c][q_i] = TKSK(q_i, λ)
14                  λ_queried[q_i] = λ
15              else
16                  R[c][q_i] = R[c][q_{i-1}]
17
18          //Extracts a set the parent trajectory of every point in R[c], ignoring duplicates
19          C_tra ← pointsToTrajectories(R[c])
20          if |C_tra| ≥ k
21              unseen_ub = calculateUnseenUpperBound() //Definition 7
22
23              //Calculates the lower bound of every trajectory in C_tra
24              seen_lb[] = calculateLowerBounds(C_tra) //Definition 8
25              descendingSort(seen_lb[]) //Sort lower bounds in descending order
26              if seen_lb[k] ≥ unseen_ub
27                  //Sort C_tra in descending order by the upper bound of every trajectory
28                  seen_ub[] = descendingSortByUpperBound(C_tra) //Definition 9
29                  foreach T_i ∈ seen_ub[]
30                      T.similarity = similarity(query, T) //Definition 5
31                      if |result| < k
32                          result.add(T)
33                      else {
34                          //result.min is the trajectory in result with the lowest similarity
35                          if similarity > result.min.similarity
36                              result.replace(result.min, T)
37                          if result.min.similarity > seen_ub[i + 1]
38                              return result
39                  break
40          λ = λ + Δ
41          c = c + 1
42      return result
```

Algorithm 1: ILA

In the original work they used the RCA algorithm by Zhang *et al.* [8] for their TkSK. We chose to implement RCA in order to emulate the original work as closely as possible. Additionally, ILA is dependent on having effective TkSK in order to perform well, as our

experiments ran on the *dif* machine described in section 6.1.1 showed that over 90% of ILA's runtime consisted of TkSK queries.

Algorithm 1 shows a pseudo-code version of ILA, the majority of which is wrapped in a `while` loop- this is the incremental part of the algorithm. For every iteration of the `while` loop (referred to as an iteration) the variable c is increased by 1, indicating which iteration we are currently at, such that at the first iteration $c = 0$. This variable is only used for tracking which iteration the algorithm is currently at. The λ variable is initialized to k, this is the minimum value that λ can be in order to answer the query given a worst case scenario where none of the query points share any terms. Note that the iteration is only continued as long as λ is less than or equal to $\lambda^{\max}$ (see definition 11). The reason for this is that when $\lambda = \lambda^{\max}$ the TkSK will have found every point that shares at least one term with any point $q_i \in Q$. Iterating any further after this point will not generate any additional candidates. We can break the rest of the algorithm down into three parts.

It should be noted that Algorithm 1 presents a slightly modified version of the original pseudo-code in [5]. During our implementation we found a number of edge cases where ILA fails to return the expected results. Algorithm 1 contains fixes for these edge cases, Appendix A describes these edge cases in detail.

Explore-and-expand: The first step is to run the TkSK to find the top-λ similar points for all query points q_i in our query Q. All retrieved points for query point q_i are added to a ranked list $R[c][q_i]$ that is sorted in descending order by their similarity to q_i. These points are then mapped to their corresponding trajectories and are collected in the set C_{tra}. Duplicate trajectories are ignored so that C_{tra} only contains one instance of every candidate trajectory. If $|C_{tra}| \geq k$ then we have enough candidates to potentially answer the query and we move on to the next step. Otherwise we increase λ by some value Δ and perform step 1 anew, starting a new iteration with an bigger λ so that we may find more candidates.

Tuning the value of Δ is an important parameter for the performance of ILA. If it is too small ILA will perform too many iterations which results in re-scanning the same points many times. On the other hand, if it is too large we risk scanning unnecessary many points. This is the problem the original authors set out to solve with GAP-ILA by using a dynamic Δ [5].

Boundary evaluation: The second step is to check if there are any unseen trajectories that might have high enough similarity to be placed in the top-k trajectories. This is done by calculating an upper boundary of similarity for all trajectories not in C_{tra} (unseen trajectories), this boundary is referred to as $unseen_ub$. Next, a lower boundary of similarity $seen_lb[]$ is computed for each trajectory in C_{tra} so that we get a list of lower boundaries and sort it in descending order. If the value of the k-th seen lower bound ($seen_lb[k]$) is higher or equal to the $unseen_ub$, we know that any unseen trajectories cannot have a similarity high enough to be placed in the top-k trajectories.
Otherwise we increase λ by some value Δ and perform step 1 again, starting a new iteration.

Aggregate results: The third and final step is to compute the upper bound of similarity for each trajectory in the candidate set C_{tra} (see section 4.1.3) and sort the set in descending order based on these bounds. Now we may iterate over each trajectory $T_i \in C_{tra}$ in order of their upper bound so that trajectories with higher upper bounds are iterated over first.

Then we compute the actual similarity $\hat{S}(Q, T_i)$ between the trajectory T_i and our query Q. If there are less than k elements in our result set, *result*, we may insert T_i into *result*. Otherwise, we check whether the similarity of T_i is higher than the trajectory with the lowest similarity (*result*.min) in *result*, in which case we replace *result*.min with T_i. Finally we check whether the similarity of $result.min$ is higher than the upper bound of similarity for the next trajectory to be scanned, T_{i+1}. If this is the case then we know that none of the future candidates to be iterated over can make it into the top-k trajectories, allowing us to stop the algorithm and return the result.

4.1.1 Upper boundary of similarity for unseen trajectories

At every iteration we can calculate the best possible similarity any unseen trajectory can have. This is used to check whether the algorithm must do another iteration by comparing it to the k-th highest lower bound; that is, the k-th candidate when the candidate set is sorted in descending order by the candidate's lower bound.

Definition 7 (Upper boundary of similarity for unseen trajectories)**.**

$$UB_{unseen}(D - C_{tra}) = \frac{\sum_{i=1}^{|Q|} \hat{S}(q_i, R_c[q_i][\lambda])}{|Q|}$$

For each unseen trajectory, no point can occur in any intermediate ranked list. So for each query point q_i, the spatial-textual similarity between q_i and a matching point (if any) in one of the unseen trajectories must be less than $\hat{S}(q_i, R[c][q_i][\lambda])$. As a result, for any unseen trajectory, the trajectory similarity between the query Q and the trajectory is less than the sum of the minimum similarity $\hat{S}(q_i, R[c][q_i][\lambda])$) [5, equation 8].

Recall definition 5 that defines point-wise similarity between two trajectories T_1 and T_2. For every point $p_i \in T_1$ we calculate the similarity between p_i and every point $p_j \in T_2$ then select the highest of these similarities as shown in definition 4. Since the ranked lists for every query point $R[c][q_i]$ are sorted by similarity, we know that $\forall p_j \in R[c][q_i]$, $\hat{S}(q_i, p_j) \geq \hat{S}(q_i, p_{j+1})$. We also know that any trajectory that is not in the candidate set has not been found by the TkSK yet. Therefore the best possible similarity for an unseen trajectory *cannot* be better than the sum of $\hat{S}(q_i, R[c][q_i][\lambda])$ for every query point $q_i \in Q$, divided by $|Q|$. Here $R[c][q_i][\lambda]$ is the λ-th point in the ranked list of similar points for query point q_i in the c-th iteration. Since the TkSK retrieves the top-λ points we know that any point not in the top-λ points must have a lower or equal similarity to q_i than $R[c][q_i][\lambda]$, $\forall p_j \notin R[c][q_i]$, $\hat{S}(q_i, p_j) \leq \hat{S}(q_i, R[c][q_i][\lambda])$.

There are cases where $|R[c][q_i]| < \lambda$, i.e in cases where the TkSK could not find λ points. This can happen when there are less than λ points that share terms with the query point. In this case we treat the λ-th point of $R[c][q_i]$ as 0. We treat it as 0 because if the TkSK could not find enough points, we know that the query point q_i will have a point-to-trajectory similarity with of 0 with any unseen trajectory T ($\hat{S}(q_i, T) = 0$), as point-to-point similarity is set to 0 when two points do not share any terms. In other words, no points in an unseen trajectory will share any terms with a query point q_i when $|R[c][q_i]| < \lambda$.

4.1.2 Lower bound of similarity for seen trajectories

The lower bound of similarity for a seen trajectory is the lowest possible similarity a trajectory can have based on the points explored so far. If this value is higher then the upper bound for all unseen trajectories (UB_{unseen}) then we know that the similarity cannot be beat by any unseen trajectory.

Definition 8 (Lower bound of similarity for seen trajectories).

$$LB_{seen}(T) = \frac{\sum_{i=1}^{|Q|} \max_{j \in [1,\lambda] \wedge R[c][q_i][j] \in T} \hat{S}(q_i, R[c][q_i][j])}{|Q|}$$

For each trajectory T which has been checked, the existing maximum similarities in all $R[c][q_i]$ can be summed and used as the lower bound of T's similarity, which is less than or equal to the real similarity because points may exist which are not in $R[c][q_i]$ [5, Equation 9].

The idea behind the lower bound is that if at least one of the ranked lists $R[c][q_i]$ contains at least one point p that belongs to some trajectory T then we know that the similarity between T and the query Q must be greater than 0. Only trajectories whose points have no common terms with any of the query points will have a similarity of 0. We also know that all of the points in the ranked list $R[c][q_i]$ have a better similarity to q_i than any point that is not a part of $R[c][q_i]$. We can therefore guarantee that the similarity between q_i and the trajectory T (point-to-trajectory similarity) cannot be less than the best point-to-point similarity between q_i and any of the points in T. Therefore the lower bound of similarity for a seen trajectory T is equal to the sum of the most similar point $p_j \in T \cap R[c][q_i]$ for each query point q_i divided by the query length $|Q|$.

Just as when computing the upper boundary for unseen trajectories (section 4.1.3), there are cases where $|R[c][q_i]| < \lambda$. In such cases we also treat $R[c][q_i][j]$ as 0.

4.1.3 Upper bound of similarity for seen trajectories

The upper bound of similarity for a seen trajectory is the highest possible similarity a trajectory can have based on the points explored so far. This value is used to sort the final candidate set in descending order; from a high upper bound to low upper bound. When iterating over the final sorted candidate set we can check if the least similar trajectory in the result set is higher than the upper bound of the next trajectory in the iteration. If it is then there is no need iterating any further, none of the subsequent trajectories will be similar enough to make it into the top-k trajectories.

Definition 9 (Upper bound of similarity for seen trajectories).

$$UB_{seen}(T) = LB_{seen}(T) + \frac{\sum_{i=1 \wedge T \cap R[c][q_i] = \emptyset}^{|Q|} \hat{S}(q_i, R[c][q_i][\lambda])}{|Q|}$$

For points in T, but not appearing in $R[c][q_i]$, their respective similarities can not be greater than the similarity of the λ-th point, $\hat{S}(q_i, R[c][q_i][\lambda])$. Thus the upper bound for T's similarity w.r.t. Q is a summation of LB_{seen} and the λ-th point's similarity [5, Equation 10].

Whilst the lower bound of similarity only considers the similarity of query points whose ranked lists $R[c][q_i]$ contains at least one point that belongs to the seen trajectory T, the upper bound only considers query points whose ranked lists *do not* contain any point that belongs to the seen trajectory T. For each ranked list $R[c][q_i]$ that does not contain any point that belongs to T we know that all of the points in T will have a lower similarity to q_i than any of the points in $R[c][q_i]$. We know this because if such a point existed, then it would already have been retrieved by the TkSK. Thus the similarity between q_i and any unseen points that belongs to T cannot be greater than the λ-th point of $R[c][q_i]$ ($R[c][q_i][\lambda]$). The lower bound covers all query points whose ranked list contain at least one point that belongs to T, and the rest of the expression covers the remaining query points whose ranked lists does not contain any points belong to T, thus covering all query points.

If $|R[c][q_i]| < \lambda$, we know that all points sharing terms with q_i have been retrieved. This means that the trajectory T cannot have any points with similarity with q_i over 0, and therefore we may treat $R[c][q_i][\lambda]$ as 0.

4.1.4 Maximum number of iterations

Recall from Definition 3 that points that do not share any terms have a similarity of 0. Therefore we can calculate the maximum number of points the TkSK will be able to retrieve for every query point. Increasing λ beyond this point will have no effect.

Definition 10 (Maximum iterations $\lambda_i^{\max}$ for point q_i). *Only a point containing at least one query term can be a candidate point in $R[c][q_i]$. Hence the maximum length $\lambda_i^{\max}$ of the ranked list $R[c][q_i]$ can be computed as follows:*

$$\lambda_i^{\max} = \sum_{j=1}^{q_i.terms} df(q_i.terms[j])$$

where df() (Document frequency) checks the total number of points that contains the term $q_i.terms[j]$.

$\lambda_i^{\max}$ is used to check whether the TkSK will be able to retrieve λ points. If not, the ranked list $R[c][q_i]$ for the query point q_i is set to $R[c-1][q_i]$, thus reusing the results from the previous iteration.

Definition 11 (Maximum iterations $\lambda^{\max}$). *A global $\lambda^{\max}$ is also set to support early termination. Early termination is defined as:*

$$\lambda^{\max} = \max_{q_i \in Q} \lambda_i^{\max}$$

There is no need to iterate further than $\lambda^{\max}$ because after this threshold the TkSK will not retrieve any additional points, thus we will not discover any further candidates.

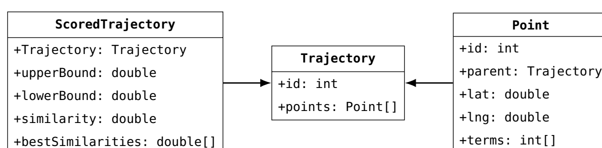


Figure 4.1: Point and ScoredTrajectory class diagrams

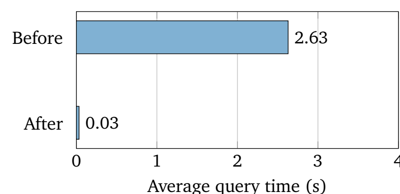


Figure 4.2: Comparison of ILA performance before and after optimizations

4.1.5 Optimizations to ILA

The original algorithm features a `R` variable that holds the points retrieved by the TkSK for every iteration. However, if we study the bounds, we find that there is no need to save the points of each iteration. To calculate all the the bounds, we only need to know three things: the number of points retrieved by the TKsK for each query point, the worst seen similarity between each query point and any point in the query point's ranked list, and the best seen similarity between each query point and every seen trajectory.

In order to track how many points have been retrieved by the TKsK for each query point we simply maintain an array with the same length as the query called `pointCounts[]` that hold the number of points retrieved. This array is indexed by the index that each query point has in the query and every time the TkSK retrieves points for a query point q_i, we update `pointCounts[i]`.

Although we do not need to save and maintain a ranked list of points for each iteration, we still need to iterate over the retrieved points and calculate the similarity between the points and their respective query point in order to know the **worst** seen similarity for each query point, and the **best** seen similarity for each candidate trajectory. In order to keep track of the worst similarity seen for each query point we maintain an array in the same vein as the `pointCounts[]` array, called `worstSimilarities[]` that saves the the worst seen similarity between each retrieved point and their respective query point.

In order to save the best seen similarity between each query point and every candidate trajectory we maintain an array called `bestSimilarities[]` for each seen trajectory, this is indexed in the same way as the previous arrays. As seen in Figure 4.1 each point p has a reference to its parent trajectory T that is used to derive candidate trajectories from the retrieved points. For each candidate trajectory we create a wrapper class called `ScoredTrajectory` that holds a reference back to the trajectory it wraps, the `bestSimilarities[]` array of the trajectory, the upper and lower bound of the trajectory, and the similarity of the trajectory. For each retrieved point p we update the `bestSimilarities[]` array contained in the parent trajectory's `ScoredTrajectory` instance, or create a `ScoredTrajectory` instance if one does not exist.

Note that the lower bound, upper bound, and similarity are only calculated once they are needed, we simply store them in the `ScoredTrajectory` class for convenience instead of maintaining three separate lists.

These optimizations significantly improve the performance of ILA since there is no need to loop through the points of each ranked list in order to find the worst and best similarities. Figure 4.2 shows the result of an experiment where we ran 100 queries on a dataset with 10000 trajectories with ILA before and after applying the optimizations described in this section. The data shows that the optimizations improve the average query times by a factor of 80.

The rest of this section describes how bounds are computed after applying the optimizations.

Optimized unseen upper bound

Algorithm 2 shows how we implemented the function for computing the upper bound of similarity for unseen trajectories with the help of the `pointCounts[]` array and the `worstSimiliarities[]` array.

```
1  function calculateUnseenUpperBound() {
2      sum = 0
3      foreach q_i ∈ Q
4          if pointCounts[q_i] ≤ λ
5              sum += worstSimilarities[q_i]
6      return sum / |Q|;
```

Algorithm 2: Unseen upper bound

Recall Definition 7 where we sum the similarities between each query point q_i and the λ-th point of the ranked list ($R[c][q_i][\lambda]$). Since the last point of the ranked list is the point with the lowest similarity to q_i we can instead simply use the value we have stored in our `worstSimilarities[]` array instead. Also, recall that if the ranked list does not contain λ points, then we treat the λ-th point of $R[c][q_i]$ as 0. Therefore we must check how many points the TkSK fetched using the `pointCounts[]` array. By adding the `if` statement we achieve the same result, as nothing is added to the sum if less than λ points are fetched.

Optimized seen lower bound

The function for computing the lower bound of similarity for a seen trajectory is very simple as shown in Algorithm 3.

```
1  function calculateSeenLowerBound(trajectory)
2      sum = 0;
3      foreach q_i ∈ Q
4          sum += trajectory.bestSimilarities[q_i]
5      return sum / |Q|
```

Algorithm 3: Seen lower bound

To calculate the lower bound of similarity between a trajectory T and the query, Definition 8 tells us to find the maximum similarity between each query point q_i and any point in its ranked list $R[c][q_i]$ that also belongs to the trajectory T. Instead of looping through the ranked lists every time we compute a lower bound we can use the `bestSimilarities[]` array of the `ScoredTrajectory` instance to find the best similarity between each query point and the trajectory. If the TkSK did not find any point belonging to the trajectory in the ranked list of q_i then the value of `trajectory.bestSimilarities[i]` is equal to 0, thus if no point that belongs to the trajectory has been retrieved by the TkSK then the lower bound will be equal to 0.

Optimized seen upper bound

To calculate the upper bound we utilize all of the arrays we have introduced in this section, as shown in Algorithm 4.

```
1 function calculateSeenUpperBound(trajectory)
2     sum = 0
3     foreach q_i ∈ Q
4         if trajectory.bestSimilarities[q_i] = 0
5             if pointCounts[q_i] ≥ λ
6                 sum += worstSimilarities[q_i]
7     return calculateSeenLowerBound(trajectory) + sum / |Q|
```

Algorithm 4: Seen upper bound

To compute the upper bound of similarity for seen trajectories we sum the worst similarity seen for each query point q_i if the TkSK has not fetched any point for q_i that belong to the trajectory. Therefore we check if `trajectory.bestSimilarities[(q_i)]` is equal to zero. We must also check whether or not there potentially are unseen points for q_i, so we check if `pointCounts[q_i]` is greater than or equal to λ, if it is, then we know that there may be unseen points that belong to the trajectory and that these unseen points cannot have a similarity to q_i greater than the worst seen similarity for q_i. Lastly we must divide the sum by the query length and add the lower bound as described in Definition 9.

4.2 2TA

Wang *et al.* [5] notes that ILA suffers from *repetitive lookups*– points explored in any given iteration will be re-scanned by the *TkSK* in all subsequent iterations. The *Two-level threshold algorithm*, referred to as 2TA, was created to solve this problem. It adopts a block-wise expansion of the search-space, much like the methods used in the RCA algorithm by Zhang *et al.* [8]. It utilizes two indexes, a grid index and an inverted index, to search for points according to spatial and textual relevance respectively. The indexes are partitioned into it_{max} blocks so that they can be incrementally searched block-by-block. The partitioning of these indexes allows for a *score-bounded expansion*, where the points explored by early iterations will have a higher similarity than those discovered later. As in ILA, we can then calculate an upper bound of similarity for any unexplored points and stop expanding the search space when it becomes apparent that no unseen trajectories can be a part of the top-k trajectories.

The textual index, referred to as the posting list, is a rank-ordered inverted index mapping terms to their respective points. The spatial index a grid-index labeled using a Z-order curve that supports range queries. These indexes are further elaborated in section 4.2.1 and section 4.2.2

```
1  function 2ta(Q, k)
2      result = []
3      it = 0
4      R = []
5      while it < it_max
6          R[it] = []
7          foreach q_i ∈ Q
8              // Initialize a ranked list for each query point per iteration
9              R[it][q_i] = []
10             R[it][q_i] ← exploreTextual(q, it)
11             R[it][q_i] ← exploreSpatial(q, it)
12         //Extracts a set of the parent trajectory of every point in R[it], ignoring duplicates
13
14         C_tra ← pointsToTrajectories(R[it])
15         if |C_tra| ≥ k
16             unseen_ub = calculateUnseenUpperBound() //Definition 12
17             //Calculates the lower bound of every trajectory in C_tra
18             seen_lb[] = calculateLowerBounds(C_tra) //Definition 8
19             //Sort lower bounds in decending order
20             decendingSort(seen_lb[])
21             if seen_lb[k] ≥ unseen_ub
22                 C_tra.clear() // Clear candidate set
23                 foreach q_i ∈ Q
24                     for p_j ∈ R[it][q_i]
25                         if Ŝ(q_i, p_j) ≥ UB_it(q_i)
26                             C_tra ← p_j.parent*)
27             Same as line 28-39 in algorithm 1
28         it++;
29     return RS;
```

Algorithm 5: 2TA

Algorithm 5 shows pseudo-code of the 2TA algorithm, like the pseudo-code for ILA, this is a slightly modified version of the original pseudo-code in [5]. It is modified to solve some edge cases that were found during our implementation, these are described in detail in Appendix A.

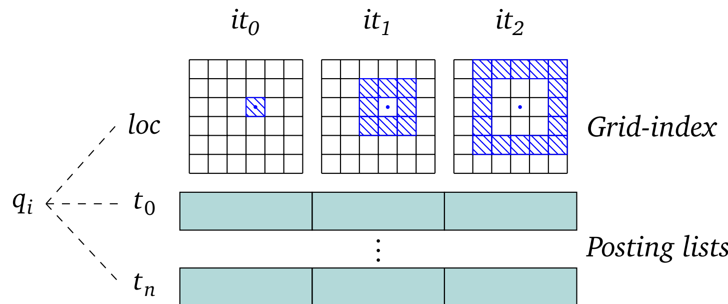


Figure 4.3: 2TA index-structure: posting list and grid-index

4.2.1 Posting list

The posting list is essentially an inverted index that maps terms to points that contain the term. A normal inverted index would simply map a term to an unordered list of points. The posting list on the other hand, maps a term to a sorted list of points ordered by the term's TF·IDF weight within each point. This list is further divided into it_{max} blocks, each of which cover an equal portion of the total TF·IDF weight-range of the term. The block-

divided sorted list for a given term t, is referred to as the posting list of term t. The total weight-range of a term t is the difference between the maximum weight $\gamma_{\max}(t)$ and the minimum weight $\gamma_{\min}(t)$. The weight-range of a block is defined as $\gamma_{range}(t) = \frac{\gamma_{\max}(t) - \gamma_{\min}(t)}{it_{max}}$. All points p that share a term t are distributed throughout the blocks of the posting list of term t based on the TF·IDF weight $\gamma(p,t)$. The block index that a point p with a term t should be placed in, is given by the function $\gamma_{block}(p,t)$.

$$\gamma_{block}(p,t) = \begin{cases} 0, \ \gamma_{\max}(t) = \gamma_{\min}(t) \\ it_{max} - 1, \ \gamma(p,t) = \gamma_{\min}(t) \\ \left\lfloor \frac{\gamma_{\max}(t) - \gamma(p,t)}{\gamma_{range}(t)} \right\rfloor, \text{ otherwise} \end{cases}$$

Table 4.1 demonstrates how a posting list is partitioned into a series of blocks for different terms.

Term	P_1	P_2	P_3	P_4	P_5	P_6	P_7
Cake	0.1	-	-	0.8	-	0.6	0.4
Sushi	0.9	-	0.3	0.4	-	0.6	0.5
Beer	-	0.8	0.2	-	0.1	-	-

Term	Block 1	Block 2	Block 3
Cake	P_4 P_6	P_7	P_1
Sushi	P_1	P_6 P_7	P_4 P_3
Beer	P_2	-	P_3 P_5

Table 4.1: Matrix of terms and the TF·IDF weight of the terms in each point (left) and a posting list with $it_{max} = 3$ constructed with the points (right).

4.2.2 Grid index

The grid index is labeled using a z-curve. It operates on similar principles to the spatial exploration performed by the RCA algorithm [8]; points are labeled and grouped using a z-curve, and stored in an array that is sorted based on the z-order label. As with RCA, exploration happens both forward and backwards in this array, and points are stored in a buffer until their proper iteration. There are however a couple of differences between this Grid Index and RCAs backing indexes, and how they are queried.

The `points` list of the `GridIndex` contains a set of sublists, each sublist being the spatial-index of a given term. The sublists are a collection of `Cell` objects, which each contain a z-label, and a list of every point that belongs to said cell (see Figure 4.4). 2TA's grid index is partitioned by keywords, much like RCA's spatial index. The difference lies in that the exploration only expands spatially, as opposed to RCA, which expands both spatially and textually. Additionally, one of the main optimizations of 2TA over ILA is that it avoids re-exploring the same points. To accomplish this, the grid index query only returns points from *the exact cells* required for the given iteration. RCA on the other hand, retrieves *all cells* up to- and including the specified iteration. This is illustrated in Figure 4.5. 2TAs grid index does this by having the grid-index maintain a state, allowing it to resume expansion when 2TA reaches its next iteration, thus never returning a point more than once.

4.2.3 Processing ETQ with 2TA

2TA begins by initializing the necessary variables before we enter the main loop. $\text{it}_{\max}$ is a parameter that defines the maximum number of iterations that are needed to explore the entire dataset. This determines the partitioning of the blocks used in the incremental expansion and is a key parameter that affects the performance of queries. It determines how

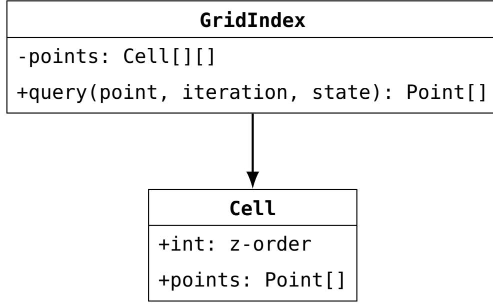


Figure 4.4: Class diagrams

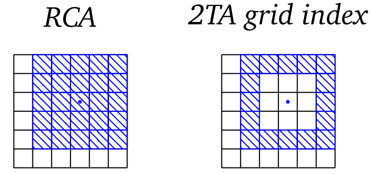


Figure 4.5: Result cells of spatial query on RCA versus 2TA's GridIndex, on $it = 3$

many blocks the posting list is split into, and the dimensions of the grid-index. In the original experiments performed in [5], this is found through a parameter sweep as it is dataset specific. They found that it_{max} is a parameter with a high impact on the performance of the algorithm.

Explore-and-expand: After entering its main loop, 2TA iterates over each query point $q_i \in$ query and uses the `exploreSpatial` and `exploreTextual` scanning operations to retrieve points in a spatial- and textual similarity bounded block. These indexes are visualized in figure 4.3. `exploreTextual` returns the it-th block of the posting list for each term, while `exploreSpatial` retrieves a set of cells with a radius of it grid-cells from the query point using the grid index. Like the ILA algorithm, these points are stored in individual ranked lists for each query point. After iterating over all query points, the ranked-lists are unified and mapped to their respective trajectories in a list C_{tra}. If the number of discovered trajectories satisfy $|C_{tra}| \geq k$, the algorithm moves to the next phase. Otherwise the iteration-counter is incremented and we restart the *explore-and-expand* phase.

Boundary evaluation: During its next phase, 2TA computes the unseen upper bound `unseen_ub` and the seen lower bound `seen_lb` for each trajectory in C_{tra} before sorting the candidates based on their `seen_lb`. If the k-th `seen_lb` in the sorted list is not greater than `unseen_ub`, we might not have found all the best candidates, and we have to increment the iteration-counter before restarting the loop. Otherwise we move on to the next phase.

Second-round-filtering: Since the candidate set may contain trajectories whose similarities are lower than the upper bound of unseen similarities we perform a second round of filtering. Definition 15 defines the upper bound of similarity for all unexplored points for a query point q_i in iteration it. There may be points in the ranked list of q_i whose similarity is less than the upper bound. These points are pruned by emptying the candidate set, before iterating through the ranked list of every query point to test if their similarity is greater than or equal to the unseen upper bound $UB_{it}(q_i)$. If so, it is added back to the candidate set.

Aggregate results: Finally, the candidate set is run through the same pruning- and aggregation routines used in the final step of the ILA algorithm, before returning the final top-k result trajectories.

4.2.4 Upper boundary of unseen trajectories

Similarly to ILA we can we can calculate the best possible similarity any unseen trajectory can have. It serves the same purpose as in ILA; checking if any unseen trajectories can have a better similarity than the k-th highest lower bound.

Definition 12 (Upper boundary of unseen trajectories).

$$UB_{unseen}(D - C_{tra}) = \alpha \cdot \sum_{q_i \in Q} UB_s(it)/|Q| + (1-\alpha) \cdot \sum_{t \in Q} UB_t(t, it)/|Q|$$

The upper bound of similarity for any unseen trajectories is calculated by summing the maximum TF·IDF weights in the relevant posting lists, and the highest possible spatial-similarity for each query point in the given iteration [5, equation 15].

For every query point q_i, any unexplored points cannot have a spatial similarity higher than the upper bound for spatial similarity as defined in Definition 14. Likewise, the textual similarity of any unexplored point cannot be higher than the upper bound for textual similarity as defined in Definition 13. Therefore we can mimic the point-wise similarity function (Definition 5) by using the same alpha and summing the spatial upper bound for every query point and summing the textual upper bound for every query point to get a upper bound of similarity for unseen trajectories.

4.2.5 Upper boundary of textual similarity

Upper boundary of textual similarity is the maximum textual relevance any unseen point can have in regards to a specific term.

Definition 13 (Upper boundary of textual similarity). *For the it-th block of term t the upper bound for textual similarity can be computed as:*

$$UB_t(t, it) = \frac{\gamma_{\max}(t) - \gamma_{\min}(t)}{it_{\max}} \cdot (it_{\max} - it - 1) + \gamma_{\min}(t)$$

Where $\gamma_{\max}(t)$ is the maximum TF·IDF of any point that shares the term t, and $\gamma_{\min}(t)$ is the minimum TF·IDF of any point that shares the term t. $it_{\max}$ determines the total amount of blocks that the posting contains [5, equation 16].

Recall that the posting lists for each term is divided into $it_{\max}$ blocks where the first block contains points with high TF·IDF weight and the last block holds points with low TF·IDF weight. At the *it*-th iteration, we will have explored every block from block 0, up to and including the *it*-th block. This means that any unexplored point cannot have a TD·IDF weight that is better than the worst point in the *it*-th block. $\frac{\gamma_{\max}(t) - \gamma_{\min}(t)}{it_{\max}}$ is the weight range of every block such that $\langle \frac{\gamma_{\max}(t) - \gamma_{\min}(t)}{it_{\max}} + \gamma_{\min}(t)\ ,\ \gamma_{\min}(t) \rangle$ is the weight interval for the last block.

By multiplying $\frac{\gamma_{\max}(t) - \gamma_{\min}(t)}{it_{\max}}$ with $(it_{\max} - it - 1)$ and adding $\gamma_{\min}(t)$, we get the lowest part of the interval of the it-th block. In the first iteration when $it = 0$ we will have explored the points in the first block, thus any unexplored points cannot have a TF·IDF weight that is better then the lowest part of the weight interval of the first block.

4.2.6 Upper boundary of spatial similarity

Upper boundary of textual similarity is the maximum spatial relevance any unseen point can have.

Definition 14 (Upper boundary of spatial similarity). *The upper bound for spatial similarity in the it-th iteration can be computed as [5, equation 17]:*

$$UB_s(it) = \frac{D_{\max} - it \cdot \frac{D_{\max}}{it_{\max}}}{D_{\max}}$$

The variable $it_{\max}$ determines the dimensions of the z-order index, and it is normalized by $D_{\max}$ so that the width and height of each cell is equal to $\frac{D_{\max}}{it_{\max}}$. When we scan the block that a query point q_i maps to, the first iteration when $it = 0$ will only scan the single cell: the origin cell. Note that we have no guarantee about where in the origin cell the query point is. In a worst case scenario the query point can be in the very corner of the origin cell, then there can be points in its very close proximity that are not scanned, whilst there may be far away points in the opposite corner that are scanned. In this scenario the spatial upper bound must and will be equal to 1 since $it = 0$. When $it = 1$ however, all the cells around the origin cell are scanned, and even if the query point is in any corner of the origin cell we know that any point within a distance of less than $\frac{D_{\max}}{it_{\max}}$ is scanned. Further when $it = 2$ we know that any point with a distance less than $2 \cdot \frac{D_{\max}}{it_{\max}}$ is scanned.

4.2.7 Upper boundary of similarity for any unprocessed points for q_i

Whilst the unseen upper bound computes the maximum possible similarity for any unseen trajectory, the upper boundary of similarity for any unprocessed points for q_i computes the maximum possible similarity any unseen point can have to a specific query point q_i

Definition 15 (Upper boundary of similarity for all unprocessed points for q_i). *The upper bound of similarity for any unexplored point for a query point q_i in the it-th iteration can be computed as [5, equation 18]:*

$$UB_{it}(q_i) = UB_s(it) + (1 - a) \cdot \sum_{t \in q_i} UB_t(t, it)$$

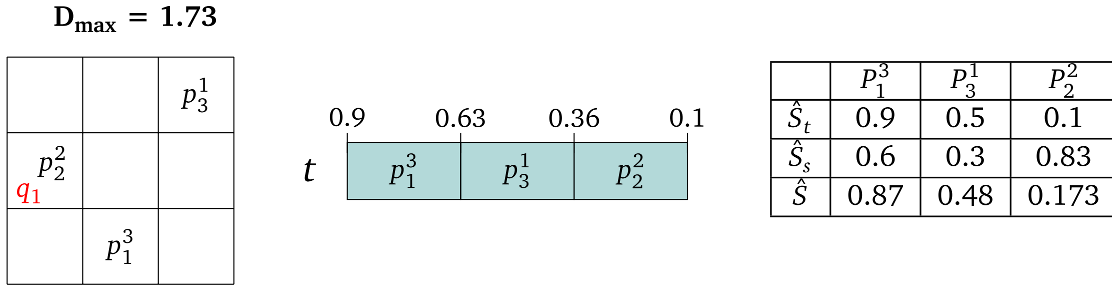
The upper bound of point-to-point similarity between a query point q_i and any unseen point p is very similar to to Definition 12. The difference is that we are calculating upper bound of point-to-point similarity for a specific query point.

4.2.8 Upper boundary of similarity for seen trajectories

Whilst the definition of seen lower bound used in ILA can be used in 2TA, the definition for seen upper bound used in ILA is not correct when applied to 2TA. Therefore we had to devise a new definition for seen upper bound.

Definition 16 (Upper boundary of similarity for seen trajectories). *The upper bound of similarity for any seen trajectory T in the it-th iteration can be computed as:*

$$UB_{seen}(T) = \frac{\sum_{i=1}^{|Q|} max(UB_{it}(q_i),\ \hat{S}_{max}(T,\ R[c][q_i]))}{|Q|}$$



	P_1^3	P_3^1	P_2^2
$\hat{S}_t$	0.9	0.5	0.1
$\hat{S}_s$	0.6	0.3	0.83
$\hat{S}$	0.87	0.48	0.173

Figure 4.6: Example where ILA's upper-bound condition fail. Displays a Grid index (left) and a posting-list (middle) for term t. The Tf-idf boundaries are shown above the posting-list. The table on the right shows textual similarity (Definition 13), spatial similarity (Definition 14) and point-to-point similarity (Definition 3) of the points.

Where $\hat{S}_{max}(T,\ R[c][q_i])$ is the maximum similarity between q_i and any point contained in $R[c][q_i]$ that also belongs to the trajectory T. If $R[c][q_i]$ contains no points that belongs to the trajectory T then $\hat{S}_{max}(T,\ R[c][q_i]) = 0$.

No definition for a seen upper bound of similarity was defined for 2TA in the original work [5]. Therefore, we assume that they use the same seen upper bound in 2TA as was specified for ILA. The seen upper bound used in ILA however, does not work when applied in 2TA. This is because Definition 9 utilizes the fact that all points in the ranked list $R[c][q_i]$ in ILA has a higher similarity to q_i, than any point not in the ranked list. This is not the case in 2TA. Unlike a TkSK, the textual and spatial indexes in 2TA do not take α into account. In cases where α is low (low spatial relevance and high textual relevance) we may find points that reside in the same grid cell as the query point, and whose similarities are worse than points which are located further away, but have better textual similarity.

Figure 4.6 demonstrates a situation where Definition 9 fails to compute a correct seen upper bound when applied in 2TA. Consider a case where $\alpha = 0.1$ and we have a query with a single point q_1 that has a single term t. In the first iteration of 2TA, two points are discovered. p_2^2 (the second point of T_2) is discovered spatially since it resides in the same grid cell as the query point, and p_1^3 (the third point of T_3) is discovered textually through the posting list of the term t. The ranked list for q_1 after the first iteration contains the two discovered points, ordered by their similarity to q_1, meaning $R[0][q_1] = [P_1^3, P_2^2]$. If we were to use Definition 9 to calculate the seen upper bound for the trajectory T_1 then we would first calculate the lower bound of T_1, which would be 0 as no points belonging to T_1 have yet been found. The rest of the expression then tells us to add the score of the λ-th point of $R[0][q_1]$, but since 2TA does not have a λ parameter we instead add the score of the lowest ranked point of the ranked list. Since $\hat{S}(q_1, p_2^2) = 0.173$, this would give us an upper bound of $UB_{seen}(T_1) = 0.173$. This is clearly incorrect since the point p_3^1 has a similarity of 0.48 which makes the actual similarity $\hat{S}(Q,T) = 0.48$. Therefore we have to devise a different and more appropriate definition.

Definition 9 makes use of the seen lower bound (Definition 8) to incorporate the query points whose ranked lists contain a point belonging to the trajectory T, whilst the rest of the expression incorporates the query points whose ranked lists do not contain any point that belongs to T. In 2TA however, a point that belongs to the trajectory T and has been

explored and inserted into the ranked list of a query point $R[c][q_i]$, can have a worse similarity than a unexplored point. We must therefore find the most similar of the points contained $R[c][q_i]$ that also belongs to the trajectory T, $\hat{S}_{max}(T,\ R[c][q_i])$, and check if its similarity is worse than the maximum similarity any unseen point can have, $UB_{it}(q_i)$. If $\hat{S}_{max}(T,\ R[c][q_i]) < UB_{it}(q_i)$ then there may be unexplored points that belong to T that can have a better similarity, and we know that these points cannot have a similarity better than $UB_{it}(q_i)$. If $R[c][q_i]$ does not contain any point that belongs to the trajectory T then $\hat{S}_{max}(T,\ R[c][q_i])$ is equal to 0. The upper bound of similarity for seen trajecotories in 2TA is therfore the sum of the maximum value of $UB_{it}(q_i)$ and $\hat{S}_{max}(T,\ R[c][q_i])$ divided by the query length.

If we use this method with the numbers in the example shown in Figure 4.6, then the seen upper bound for the trajectory T_1 after the first iteration will be $UB_{seen}(T_1) = 0.667$. The seen upper bound is required to never underestimate the actual similarity. This method will never underestimate the actual similarity of trajectories.

4.2.9 Optimizations to 2TA

We utilize the same optimization method in 2TA as in ILA using the `ScoredTrajectory` class to track the best seen similarities for each candidate trajectory. We do not have use for the `pointCounts[]` array or the `worstSimilarities` array that is used in the ILA. Like in ILA this alleviates the need to keep ranked lists for each query point and makes seen lower- and upper bound computations significantly faster since we don't need to loop through all of the ranked lists.

The second round of filtering in 2TA calls for us to loop through the ranked list of each query point q_i and compare every point with the upper bound of similarity for all unprocessed points for q_i (Definition 15). Note however, that if any point in the ranked list of q_i passes the second round filtering, then the parent trajectory is not pruned. Only trajectories whose points all fail the second round filtering are pruned. The `bestSimilarities[]` array in a trajectory's `ScoredTrajectory` instance holds the best similarity of the most similar point seen for every query point. Instead of looping through the ranked list of each query point, we can simply loop through all the candidates and check if any of the similarities stored in the trajectory's `bestSimilarities[]` array is higher than the upper bound of similarity for all unprocessed points for the corresponding query point q_i. This accomplishes the same as if we looped through the actual points, but requiring significantly fewer iterations and calculations.

For 2TA we did an optimization that we did not do for ILA, an optimized k-th lower bound computation. On line 20, Algorithm 5 calls for a complete sort of the computed set of seen lower bound values for all candidates in C_{tra}. However, the lower bound of a trajectory T only changes if a seen point has a better similarity to q_i than any previously explored point. In order to optimize the computation of the k-th highest lower bound we keep a persistant min-heap of k size throughout all iterations of a query. In the first iteration the min-heap is empty, so we will have to calculate the seen lower bound of all trajectories discovered in the first iteration. In subsequent iterations however we only need to compute the seen lower bound of trajectories whose `bestSimilarities[]` array were updated. For every candidate trajectory whose seen lower bound was updated, we only need to check if the head of the min-heap is lower than the newly computed lower bound for each trajectory.

If it is, we have to insert it into the min-heap, otherwise we can ignore it. In some cases a trajectory with an updated seen lower bound can already be present in the min-heap. In those cases we simply update the existing seen lower bound. This reduces the cost of calculating the k-th highest seen lower bound in two ways; we don't need to compute the seen lower bound for trajectories whose seen lower bound has not changed, and we don't need to sort the entire candidate set by their seen lower bound in every iteration. This reduces the complexity of the k-th highest lower bound computation from $O(n \log k)$, where n is the total number of candidates in each iteration and k is the number of requested results (the size of the min-heap), to $O(m \log k)$ where m is the number of trajectories whose seen lower bound was updated in each iteration.

The computation of Definition 12, Definition 13, Definition 14, and Definition 15 are trivial and have not been optimized in any way and they are therefore not explicitly described.

Optimized seen lower bound

Algorithm 6 show how the seen upper bound for 2TA was implemented using the `bestSimilarities[]` array of a candidate trajectory.

```
1 function calculateSeenLowerBound(trajectory)
2     sum = 0;
3     foreach q_i ∈ Q
4         sum += max(trajectory.bestSimilarities[q_i], calculateUnseenUpperBoundForQueryPoint(q_i))
5     return sum / |Q|
```

Algorithm 6: Seen lower bound

The `calculateUnseenUpperBoundForQueryPoint(`q_i`)` function computes the upper boundary of similarity for all unprocessed points for q_i. As in the optimization for `seen_lb` of ILA, we use the `bestSimilarities[]` array in order to avoid iterating over the entire ranked list.

5. Parallel ETQ processing

This chapter will introduce a parallel implementation of the 2TA algorithm, referred to as 2TAP, and our 2TA-inspired algorithm for applying distributed computing to processing ETQs: *Ellsworth*. Both of these are intended to be able to process exemplar trajectory queries at larger scale than demonstrated by Wang *et al.* [5], though in the case of Ellsworth, a much larger scale.

5.1 2TAP: Multithreaded 2TA

There are two primary motivations behind 2TAP. One motivation is to push the performance of 2TA on a single machine, by utilizing more of its available CPU resources without making any major changes to the overall algorithm. Another motivation is to provide a more fair baseline in order to compare single-machine ETQ processors, versus the cluster-based *Ellsworth*. In order to adapt the Two-Threshold algorithm to utilize the capacity of a multi-core system, we've parallelized the most computationally demanding phase of 2TA. Through profiling the code, this was found to be the *explore-and-expand* (lines 7 - 14) phase. The goal of this optimization is then to allow exploration to happen in a parallel fashion, while still maintaining thread-safety when compiling the results for the next phase of the algorithm.

In our original 2TA implementation, as soon as a point is explored by either spatial-expansion or textual-expansion, both the point's `bestSimilarities` are updated and its parent `ScoredTrajectory` is updated. In the parallelized 2TA, the exploration is split into a series of subtasks that are executed in parallel. Normally 2TA would iterate over each query point q every iteration, and run `exploreSpatial` and `exploreTextual` sequentially. These are instead run in parallel, as $|Q| * 2$ subtasks, as shown in Algorithm 7. However, to maintain thread safety, points are explored but not expanded immediately. The discovered points are placed into a bucketed buffer, and not expanded until all the exploration tasks are complete. The number of buckets is set to equal the number of threads available to the system, and the bucket-index for each point is determined based on the point's trajectory id. When the exploration tasks are finished, a thread is spawned for each bucket, and the buffer is processed in parallel. As the points are bucketed based on their parent trajectory id, each `ScoredTrajectory` will never be processed by multiple threads simultaneously, thus preventing the use of expensive locking mechanisms to avoid race-conditions.

```
1         parallel for q_i in Query:
2             for point in exploreTextual(q_i, it)
3                 bucket = point.parent.id % numThreads
4                 buckets[bucket].add(point)
5         parallel for q_i in Query:
6             for point in exploreSpatial(q_i, it)
7                 bucket = point.parent.id % numThreads
8                 buckets[bucket].add(point)
9         await //wait for all parallel tasks to complete
10        for thread in range(0, numThreads):
11            start_thread(expand_points(bucket[thread]))
```

Algorithm 7: Point processing in 2TAP

5.2 Ellsworth: ETQ on Spark

We adapted the 2TA algorithm from section 4.2 to use Apache Spark in order to create an ETQ processor named Ellsworth, which can run on a cluster. It utilizes the same score-bounded exploration of the dataset and the same *second-round filtering* process, but does not use a grid index or a posting list index. Spark does not easily permit the use of local persistent indexes, as it targeted at for batch-processing. Therefore RDDs do not provide random access to individual values. In order to explore the dataset, Ellsworth must therefore iterate over all the points in the dataset to select the desired points. The points of the dataset are partitioned throughout all of the cluster. For each partition of the dataset, Ellsworth performs a partition local top-k search which is aggregated in the end to find the true top-k trajectories.

A key consideration when designing a distributed query system is partitioning data into subsets which are processed on the different nodes. Failing to distribute the data in line with the computations will lead to increased need for data shuffling later, which is typically one of the most expensive operations on such a system, and can lead to performance bottlenecks [23]. It is therefore important to partition the data efficiently to reduce the amount of data the nodes have to shuffle between themselves during queries. Hence, a fundamental part of Ellsworth is how the points and trajectories are partitioned. Recall the *ScoredTrajectory* class that hold the best score that has been seen between any point in the trajectory and each query point. These similarities are required to calculate the lower and upper bounds. In order to find the best similarities we need to maintain a single instance of *ScoredTrajectory* for each trajectory that is discovered during point exploration. This is a problem if points are randomly distributed across the nodes, as it would require the data to be shuffled across the nodes in every iteration. Avoiding this was a key consideration when adapting 2TA to run on Spark and it is the primary concern behind the partitioning scheme of Ellsworth.

5.2.1 Partitioning scheme

Ellsworth uses two *working datasets* to perform queries, an RDD of points called the *pointRDD*, and an RDD of trajectories, called the *trajectoryRDD*. Both RDDs use the same partitioner; the trajectories are partititioned by the id of the trajectory, and the points are partitioned according to the id of their parent trajectory. This ensures that all points with the same parent trajectory are placed in the same partition and that all parent trajectories of one partition of the *pointRDD* reside in exactly one partition of the *trajectoryRDD*. For example,

if a partition of the *pointRDD* contains two points, then the parent trajectories of these two points will reside in the same partition of the *trajectoryRDD*. In other words, the two RDDs are co-partitioned.

The points are initially loaded from HDFS into an RDD which is partitioned according to the HDFS blocks. Next we execute a *map* transformation on the RDD to map it from a RDD of points, to a key-value RDD. Here the parent trajectory id acts as a key pointing to a single point (RDD<Point> → RDD<ParentId, Point>). The parent id is an integer, so in order to partition the points we simply calculate a partition key pk from the parent id $p.parent$ and the desired number of partitions n by performing a modulus operation $p_{tid} \equiv pk \pmod{n}$. We create a partitioner using this method and use the *repartition* operation to re-partition the points and trajectories according to their new partition keys to create the *pointRDD* and *trajectoryRDD*.

5.2.2 Query execution

During initialization of Ellsworth, the points and trajectories are read from HDFS and re-partitioned to create the *pointRDD* and *trajectoryRDD*. These RDDs are re-used across queries. Unlike 2TA, Ellsworth does not use a true posting list or a grid index, as it must iterate over the *pointRDD* to select points that are relevant to the current iteration. Therefore, we still need to be able to determine which grid cell a point belongs in and which posting list block the terms of each point belongs in. In order to calculate this we compute both the maximum and minimum TF·IDF weights for each term, and the minimum longitude and latitude of any point. Figure 5.1 shows an overview of how queries are executed in Ellsworth. We can break down the query execution into six different steps.

Dataset exploration: The first step is exploration. Not every point is relevant in every iteration, and many points are explored multiple times by different query points. In each iteration we iterate over the *pointRDD* and perform a *mapValues* transformation. This transforms the value in each key-value pair to a new key-value pair with the same key but with a new value. Each value is mapped to a tuple consisting of the point and a set of query points that the point is relevant to, or null if the point is not relevant at all. Then a *filter* transformation is performed to filter away all of the key-value pairs where the value is null, thus filtering away points that are not relevant to the current iteration. We call the resulting RDD of explored points for the *exploredRDD*. Both the *map* and *filter* transformations preserves partitioning and thus have the same partitioning as the *pointRDD*.

Candidate expansion: The next step is to get a candidate set from the explored points. Due to the partitioning scheme we know that points that belong to the same trajectory are located in the same partition. We can therefore perform a *mapPartitions* transformation on the *exploredRDD* in order to create a `ScoredTrajectory` instance for every candidate in every partition. For each partition we create a set of `ScoredTrajectory` instances for each discovered candidate and update its `bestSimilarities[]` array as we process the points in the partition. This gives us a new key-value RDD where the id of the candidate trajectory is the key and the candidate's `ScoredTrajectory` instance is the value. This RDD is called the *newCandidatesRDD* and has the same partitioning as the *exploredRDD*. If we are in the first iteration then we persist this RDD throughout the entire query and name it the *candidatesRDD*. If we are in any subsequent iteration, we now have two RDDs, the *candidatesRDD* from the previous iteration and the *newCandidatesRDD*. These candidates are from two completely

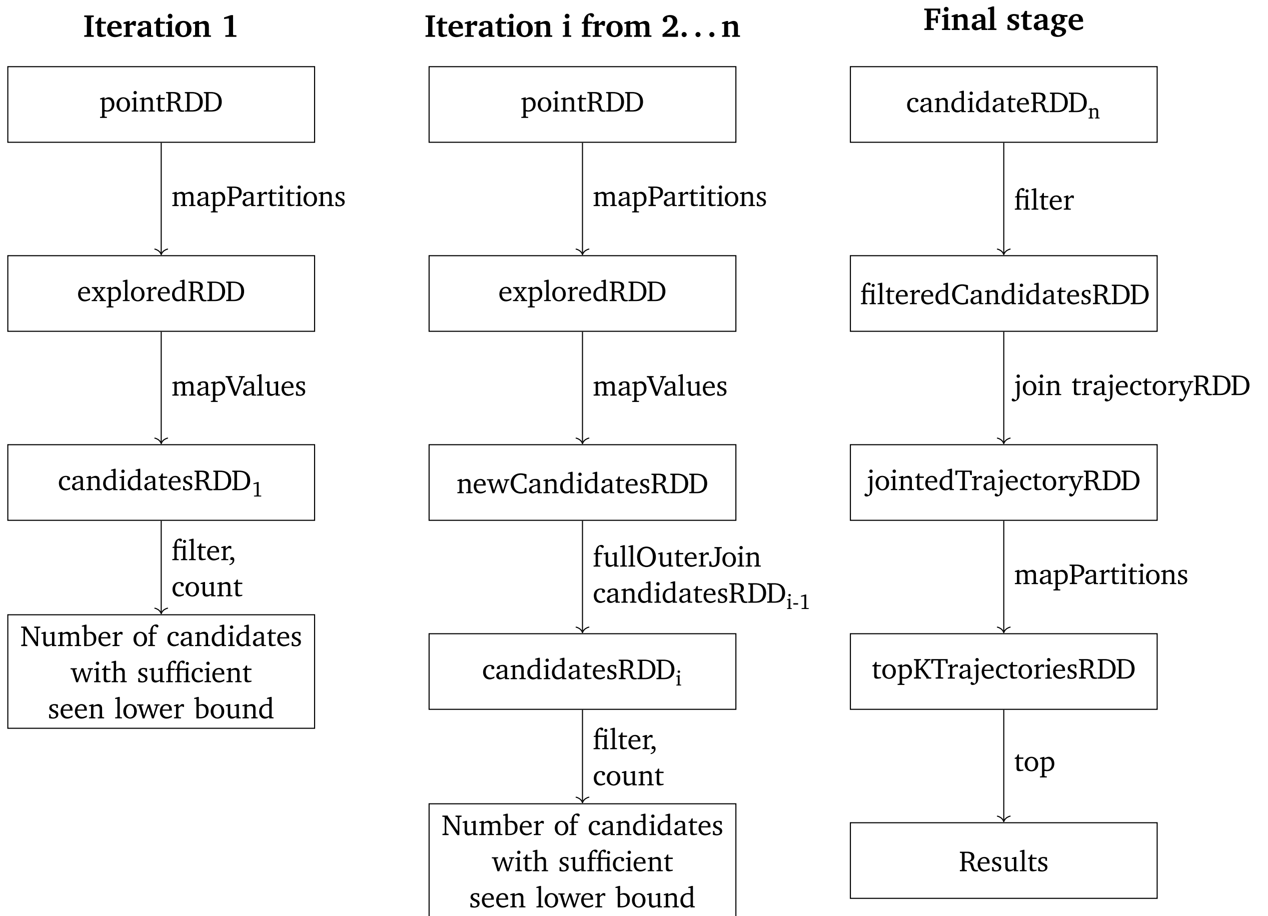


Figure 5.1: Overview of query execution in Ellsworth

disjointed sets of points, but can still contain the same candidate trajectories. In order to make sure we only ever keep a single instance of `ScoredTrajectory` for each candidate we have to join and merge the *candidatesRDD* and the *newCandidatesRDD*. We do this using a *fullOuterJoin* operation. This operation performs a full outer join of two RDD based on their keys. In cases where both the RDDs contain the same key, we must merge the two `ScoredTrajectory` instances to form a single instance. This is done by simply iterating over the `bestSimilarities[]` array of each instance and selecting the best value for each element. Normally, a join like this would require Spark to perform a data shuffle. However, since the RDDs are co-partitioned, Spark can perform a co-partitioned join, which is significantly less expensive.

Candidate counting: In the third step the *candidateRDD* contains every candidate seen so far. In order to determine if another round of exploration is necessary, we calculate the `seen_lb[]` of each trajectory and count how many of these are higher than the `unseen_ub` for the current iteration. This is done by simply performing a *filter* transformation to filter away any candidates with insufficient `seen_lb` and finally executing a *count* action to determine whether we have with at least k candidates.

Second-round filtering: At the fourth step we know we have enough candidates to answer the query. Like in 2TA we perform a second round of filtering to further reduce the amount of candidates. This is done the same way as described in section 4.2.9, by iterating over the `bestSimilarities[]` array of each `ScoredTrajecory` and checking if `bestSimilarities[`q_i`]` is better than $UB_{it}(q_i)$ (Definition 15). This is done by performing a *filter* transformation and creates an RDD called the *filteredCanidatesRDD*.

Partition local top-k: In order to be able to calculate the similarity between the query and every candidate, we need to join the *trajectoryRDD* with the *filteredCandidatesRDD*. Both RDDs are key-value RDDs with the trajectory id as key. As both use the same partitioner, Spark will perform a co-partitioned inner join. The resulting RDD will be another key-value RDD with the trajectory id as a key and the value being a tuple consisting of the trajectory itself, and its `seen_ub`. Next, we perform a *mapPartitions* transformation to do a partition local top-k search. This is done the same way as in 2TA where we iterate over the candidates in descending order based in the candidate's seen upper bound. If the upper bound of the next candidate to be iterated over has a seen upper bound less than the similarity of the worst candidate in the top-k set, we know that any subsequent trajectories cannot make it into the top-k.

Top-k: At this point each partition consists of k trajectories. The next step is to collect the top-k trajectories from all partitions into a single top-k set. This is easily done by using the *top* action.

In short, Ellsworth aims to answer top-k exemplar trajectory queries using a Spark cluster. This is done by emulating the 2TA algorithm; parallel incremental exploration of both spatial- and textual space and pruning away points as early as possible. In order to do this in an efficient and distributed manner, the partitioning of data is essential to avoid unnecessary shuffle.

5.3 Naive ETQs on Spark

We also created a naive ETQ processor on top of the Apache Spark framework in order to test if Ellsworth performs better than a simple naive method. It only uses a RDD of trajectories. It calculates the point-wise similarity between all trajectories and the query by running a *map* transformation to transform each trajectory to a key-value pair where the key is the trajectory, and the value is the similarity. Then we use the *top* action to collect the top-k trajectories based on the value of the key-value RDD.

6. Experiments

This chapter will deal with the experiments we conducted to evaluate the different ETQ implementations. Here we will also detail the environment used to run the experiments, as well as which datasets were used. Our experiments are split into three phases:

- **Phase 1:** Comparing our implementations of the algorithms by Wang *et al.*: 2TA and ILA. We also include a very naive ETQ solver in order to establish a baseline. The purpose of this phase of experiments is to verify the performance of these algorithms, and to compare these findings to the results from [5].
- **Phase 2:** Testing how the algorithms can perform on larger sets of data. We will also apply some multithreading to 2TA, in order to better utilize the capabilities of a multicore system, while the general algorithm remains the same. These tests will be run on a single computer, and will serve as a point of comparison for the distributed implementation of ETQ.
- **Phase 3:** Testing the viability of our own distributed ETQ processor, Ellsworth. For these tests we will use the *dascosa* cluster, which is detailed in the next section. These experiments will be used to answer RQ3.

6.1 Setup

The experiments are all run with a series of synthetic datasets based on a real Foursquare dataset from the New York metropolitan area. Using this source-dataset of about 1000 trajectories and a total of 200 000 points, we generated a series of datasets of varying sizes, ranging from 10k trajectories to 200 million trajectories, in order to test how the different algorithms performed at different dataset sizes. The experiments are run in two different environments: a single machine named *dif*, and a Spark cluster named *dascosa*. The first two phases of experimentation are run on the *dif* machine, while the third and final phase experiments run on the *dascosa* cluster. All experiment results are averaged by running either 100 or 1000 queries, this is specified below the graph of each experiment. Note that when measuring average run time, we only measure the time from a query is submitted, until the query result is ready. Time spent building indexes, or materializing RDDs in the case of Ellsworth, are not taken into account.

6.1.1 Environments

dascosa: This is a set of computes consisting of 25 nodes that collectively form a cluster, which runs various services. The cluster provides data storage through Apache HDFS, and Yarn backed by Zookeeper for resource management and task scheduling. The cluster also runs Spark, which is our primary interest. Not all nodes have the same specifications or

<table>
<tr><th>dascosa03 - 16</th><th>dascosa17 - 25</th></tr>
<tr><td>CPU: Intel(R) Xeon(R) CPU E5-2640 v3
Cores: 16 (32 virtual)
CPU Frequency: 2.60GHz
Memory: 128GB</td><td>CPU: Intel(R) Xeon(R) Silver 4210
Cores: 20 (40 virtual)
CPU Frequency: 2.20GHz
Memory: 192GB</td></tr>
<tr><td colspan="2">OS: Ubuntu 18.04
JVM: java-11-openjdk-amd64
Cloudera 6.3.1, Spark 2.4.0</td></tr>
</table>

Table 6.1: Specifications of the *dascosa*-cluster nodes

dif
CPU: Intel(R) Xeon(R) Gold 5118 Cores: 24 (48 virtual) CPU Frequency: 2.3GHz Memory: 384GB
OS: Ubuntu 18.04 JVM: java-11-openjdk-amd64

Table 6.2: Specifications of the *dif* machine

perform the same tasks: the first two nodes (*dascosa01* and *dascosa02*) perform some administrative tasks within the cluster, such as load balancing and act the Yarn master. The remaining 23 nodes act as workers. The worker-nodes' specifications are listed in Table 6.1.

dif: This is a single-machine with a fair bit of memory and computing power. *dif* runs Ubuntu, powered by two Intel Xenon Gold 5118 Processors, and a total of 384GB memory.

6.1.2 Datasets

As mentioned, all the datasets used for our experiments are synthesized from a Foursquare set, referred to as the NYC set. We created our own dataset generator in order to be able to create datasets of varying sizes, but also to be able to examine the impact of different parameters, such as trajectory length, or the number of terms per point. When generating the datasets used for the experiments, we set the generator parameters to echo the properties of the NYC dataset used by Wang *et al.*, making our results more comparable to the original results.

By repeating a cycle of sampling various data, such as point locations, term distributions and trajectory lengths whilst adding some noise, we can generate several millions of trajectories. Query sets are created in the same manner. A heatmap of such a dataset is shown below in figure 6.1

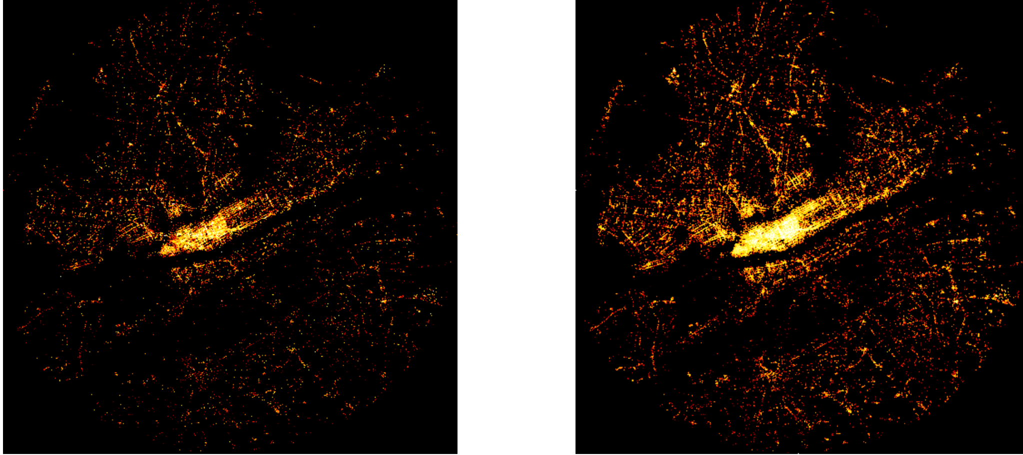


Figure 6.1: Source NYC dataset on the left, synthetic 200 000 trajectory dataset on the right

6.2 Comparing ETQs

As shown in chapter 4, we've created our own implementations of the algorithms 2TA and ILA. In order to answer RQ1, we also need to reproduce similar experiments to investigate how our implementations hold up against the originals. We will be testing how the algorithms perform using (1) different values of max iterations it_{max}, (2) various α weights, (3) size of the query trajectory $|Q|$, (4) varying number of terms per point of the query $|q.term|$ and (5) desired number of results k. When running these experiments, we will be using the *dif* machine and a synthetic dataset *NYC-49k* which mimics the properties of the dataset used in the original paper. As mentioned in section 6.1.2, this is a set from the New York metropolitan area, and it consists of about 50k trajectories. Table 6.3 details more of the properties of this dataset. Note that the dataset only mimics the properties listed in [5], so it is by no means an exact match. Therefore, properties such as spatial distribution of points and distances between points in trajectories are likely to mismatch, and impact the performance of the algorithms in various ways. The query-set is generated in the same manner as the NYC-49k dataset, although using different parameters. These are shown in Table 6.4.

Before running tests on ILA, we performed a parameter-sweep in order to find a suitable Δ value. In the original work, a Δ value of 1000 was used, but our tests revealed that a Δ of 10000 yields a 3x performance increase on the NYC-49k dataset. Therefore, $\Delta = 10000$ is used for all the following experiments.

$\mathbf{it_{max}}$: First, we run the algorithms on different values for it_{max}. Preliminary testing has shown that the choice of this value has a high impact on the performance of the algorithms. Initially, we ran these tests with the same it_{max} range (10–190), as in the source paper as shown in 6.2b. We observed that the graph suggested that it_{max} values below 10 may give us better performance, so we ran an additional test on the range it_{max} 1–10 as shown in Figure 6.2a. We found that an it_{max} value of around 3 yields the best performance on both ILA and 2TA. This is a drastically different result compared to the results in the original work. Wang *et al.* found that a it_{max} value near 150 iterations was optimal for both ILA and 2TA, whereas our tests suggested that lower it_{max} performs far better. In Figure 6.2b we can see that the average query duration on ILA increased linearly as it_{max} increased, while

NYC-49K	
Number of trajectories	49 027
Number of points	204 960
Mean Trajectory length	4.18
Mean trajectory distance	4 858m
Mean term count	9.96
$D_{\max}$	49 794m
Terms in corpus	19 146

Table 6.3: NYC-49k synthetic dataset

Queries	
Number of trajectories	1000
Number of points	10 000
Query length	10
Terms per query point	5

Table 6.4: Set of query trajectories used for tests

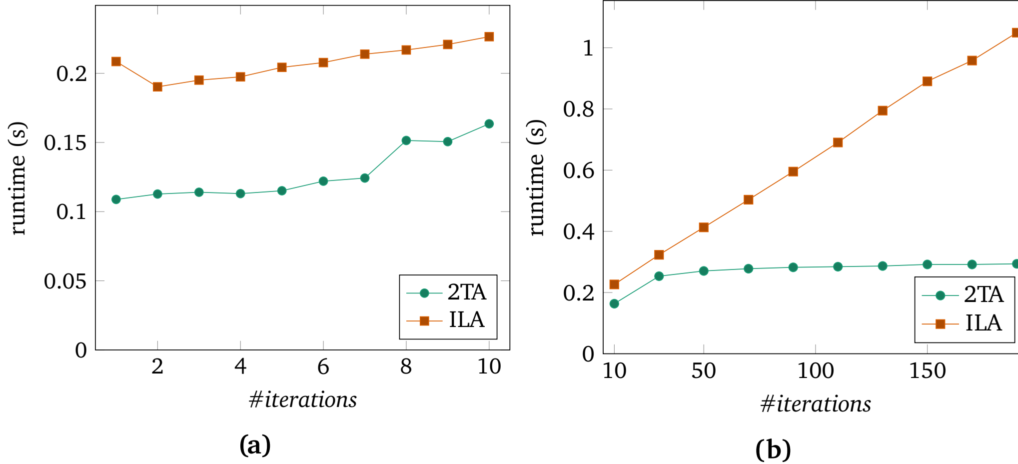


Figure 6.2: Results of it_{max} tests on two ranges of it_{max}, 1 - 10 (left), and 10 - 190 (right). N = 1000.

the optimal value being around 3 iterations as seen in Figure 6.2a. Note that there were some very minor fluctuations in different test-runs, but the optimal still remained around 3. Running queries on ILA at $it_{max} = 10$ is approximately 3x faster than on $it_{max} = 150$. The difference in performance for 2TA is not nearly as dramatic, but still shows a monotonically increasing querying time as it_{max} increases. This is likely due to our system being a purely in-memory implementation, and so we are not as heavily penalized when performing random access on a point. Seeing that we do not incur any additional disk IO when accessing a point, it seems to be cheaper to access- and expand more points, rather than performing several boundary-computations in every iteration. An it_{max} of 3 will be used for the remaining experiments of this section, both for ILA and 2TA.

alpha: Next, we examine alpha values, using the best-performing it_{max} values discovered in the previous test. The tests are run using α values from 0.1 to 0.9, at increments of 0.1. We can observe a trend where ILA performs better on higher α values where textual similarity is emphasized, whereas 2TA is fairly even across all values.
k: Here we vary the number of results requested for each query. We expect to see a relatively flat graph for both 2TA and ILA, as was observed by Wang *et al.* This is confirmed, as shown in figure 6.3b.

|Q|: For this test, we've generated several new sets of queries, with a set number of

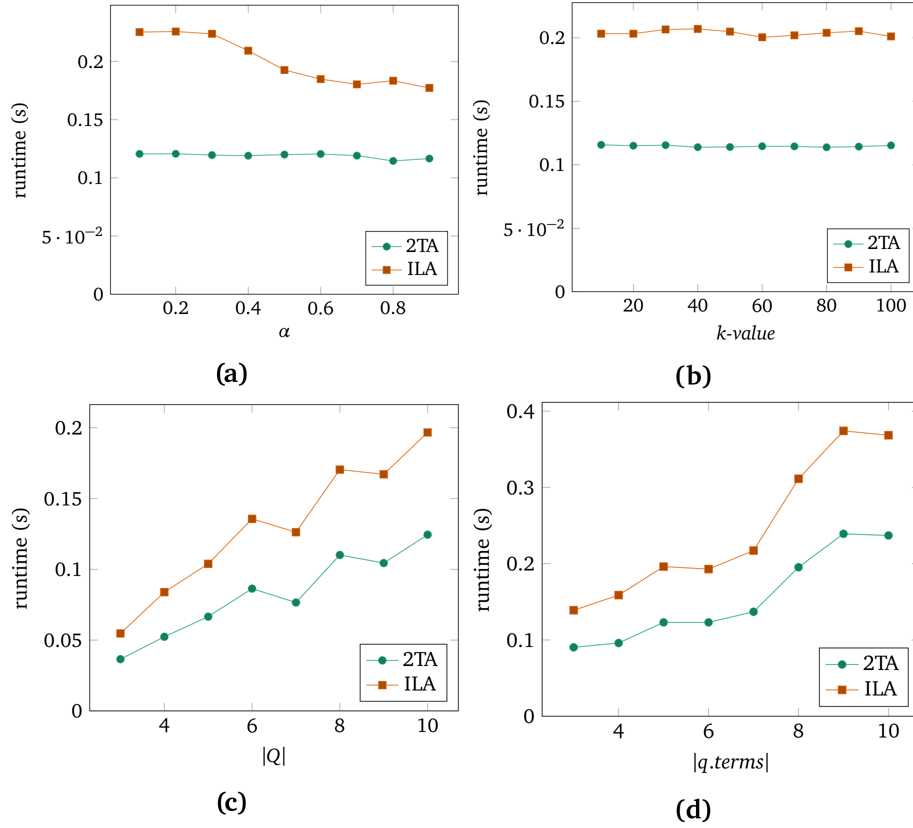


Figure 6.3: Average query time on ILA versus 2TA using various parameters tests on NYC-49k. N = 1000.

points per query. Otherwise the query sets are similar to the ones used for the previous experiments. The graph shows a trend for both 2TA and ILA where higher values of $|Q|$ (i.e. longer query trajectories) requires more time to process.

|q.*terms*|: Like the previous test, we've generated a new series of queries for this experiment. In this case, we have set the number of terms per query point q in the query trajectory Q. The number of terms range from 3 - 10. Again, we observe a trend where a higher number of terms increase the time required to solve a query. This echoes the trends observed by Wang *et al.*

Finally, we implemented a naive ETQ solver, which simply loops over all trajectories in the dataset and computes their similarity to the query trajectory, $\hat{S}(Q, T)$, before sorting the list and returning the top k results. The purpose of this is to establish a baseline, in order to determine what the actual performance gains of using an algorithm like ILA or 2TA are. This experiment uses the same setup as previous tests; NYC-49k dataset, 1000 queries, $\alpha = 0.5$, and an it_{max} of 3 for both ILA and 2TA. The result of this test are presented in Figure 6.4 and confirms that both ILA and 2TA are much better than a naive approach.

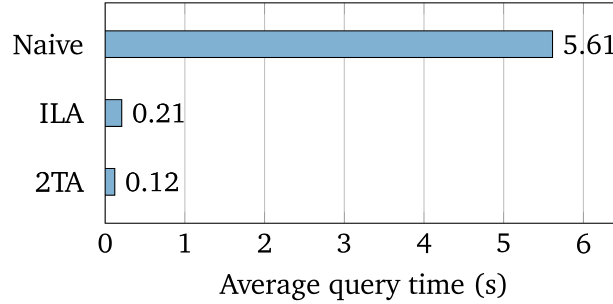


Figure 6.4: Comparison of 2TA and ILA with a Naive baseline. N = 1000.

6.3 Scaling ETQ

The results of Wang *et al.* as well as our results from the previous section has already demonstrated that these algorithms are capable or processing exemplar trajectory queries within a reasonable time. But if we consider the volumes of data that are available today, a dataset of 50 000 trajectories consisting of about 200 000 points, is not particularly large. RQ2 poses the question of how these algorithms scale with larger set of data. In this part of the experimentation we will investigate how ILA and 2TA perform when the size of the datasets grow, and attempt to push the performance of 2TA by applying multithreading. For the following tests, we have synthesised datasets consisting of 10 000 trajectories, all the way up to 1 900 000 trajectories. All datasets are shown in table 6.9, while the query set remains the same as in Table 6.4. All the synthesised datasets still use the Foursquare New York metropolitan area as source dataset.

Figure 6.5 show the result of the scalability test on 2TA and ILA. It clearly shows that 2TA is better suited for processing larger volumes of data than ILA. We can observe some oddities where a larger dataset leads to a improved performance, such as at NYC-1000k and NYC-1800k for ILA and NYC-1500k for 2TA. These are likely artifacts due to the randomness of the generated datasets, and possibly other processes competing for resources on the host machine, however the general trends remain. On the NYC-10k dataset, ILA spent an average of 60ms on each query, while 2TA used 20ms. When the dataset size increased to 1 000 000 (NYC-1000k), 2TA still only needed 3.9 seconds per query which is 150 times more than on the 10k dataset. ILA on the other hand required 26.9 seconds, which is nearly 450 times slower than its performance on NYC-10k.

Extrapolating off of this, if we were to process a larger dataset of 100 million trajectories, 2TA would process a query in about 3 minutes, while ILA would require (assuming a linear scaling) a whole hour to complete a query. If we were to expand this even further to a dataset of 1 billion trajectories, still assuming a linear scaling of performance, 2TA would need about 32 minutes and ILA nearly 10 hours. Note that at these sizes our implementation would require vastly more memory, and therefore require either a larger bank of memory or rely on disk access, which would further increase querying-time.

In order to attempt to improve on the performance of 2TA, and better utilize the available resources on the *dif* machine, we implemented 2TAP. This is a somewhat modified 2TA, with multithreading abilities, as detailed in section 5.1. To test the achieved speedup of using 2TAP, we ran two query-sets through 2TAP, using a different number of threads.

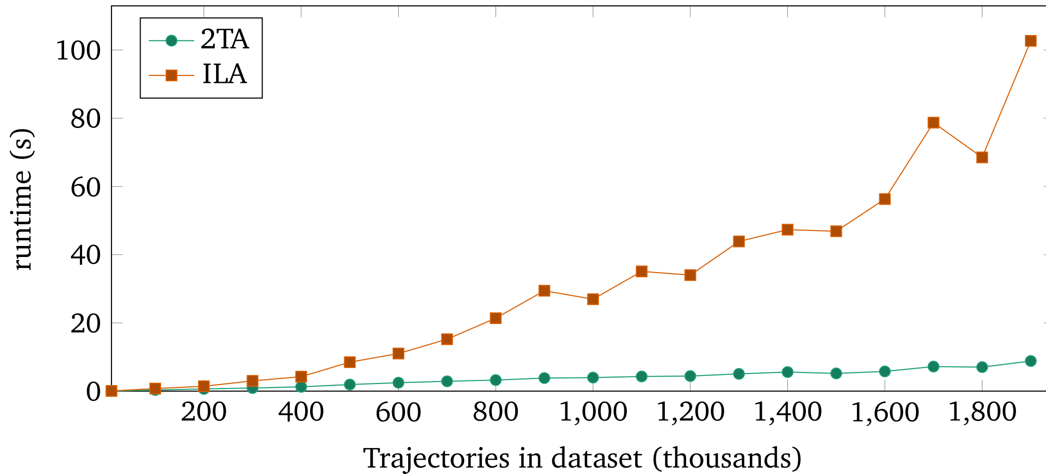
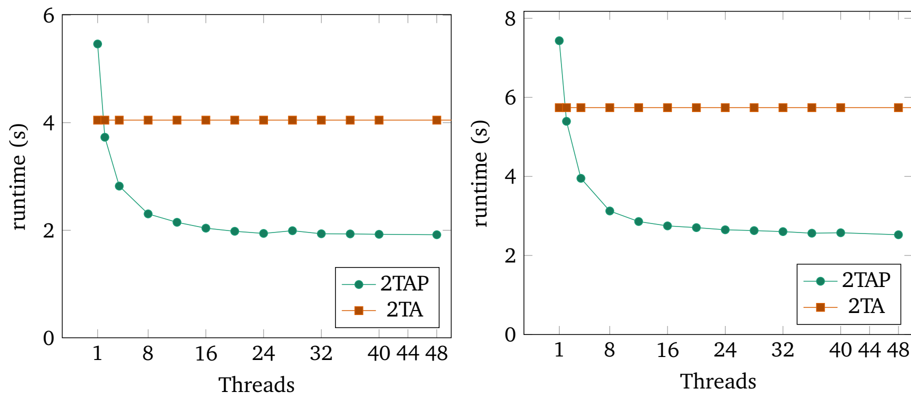


Figure 6.5: Results of the scalability test, comparing ILA to 2TA on a range of large datasets. N = 1000.



(a) Standard of queries where $|Q| = 5$ **(b)** Set of queries where $|Q| = 20$

Figure 6.6: Results of parallelism experiment. N = 100.

The first query-set is the same as used previously (Table 6.4), while the other has longer trajectories, as we set $|Q| = 20$. Otherwise they are identical. The graphs also include a run of 2TA, as a baseline.

The results indicate only a 2x speedup, despite using 48 threads rather than a single thread. There are several shortcomings of 2TAP which may be the cause of this. We discuss these further in Chapter 7. For the next experiment, we ran 2TAP through the same tests as we subjected ILA and 2TA to, as described above. We used the same datasets ranging from 10k - 1.9m trajectories as well as the same queries, and compared its performance to the normal 2TA. Results are shown in figure 6.7.

In figure 6.7 we can observe that 2TAP runs faster than 2TA, as expected. Compared to 2TA, which had an average query-time increase of 150 from NYC-10k to NYC-100k, 2TAP only suffered a 50 times query-time increase. This demonstrates that the 2TA can be extended to support parallel processing with relatively few modifications, and achieve a significant performance boost. However, regardless of faster querying-times, 2TAP will still

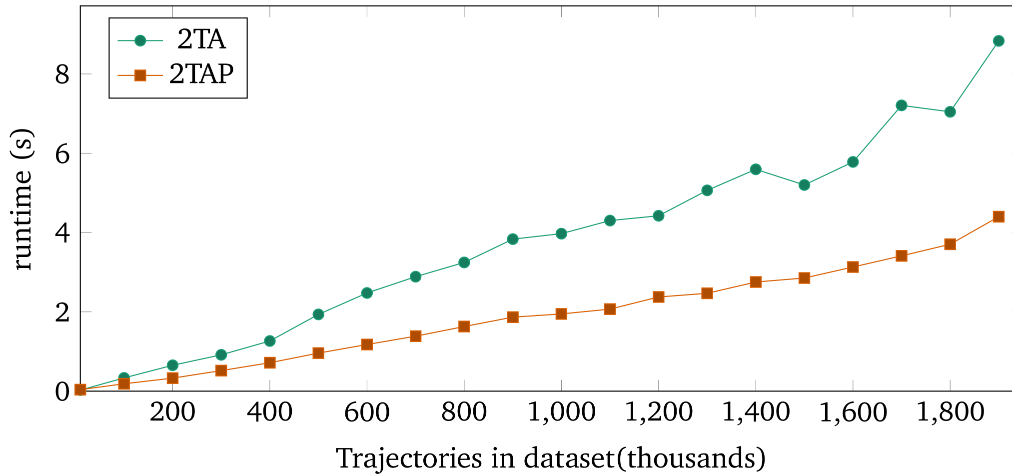
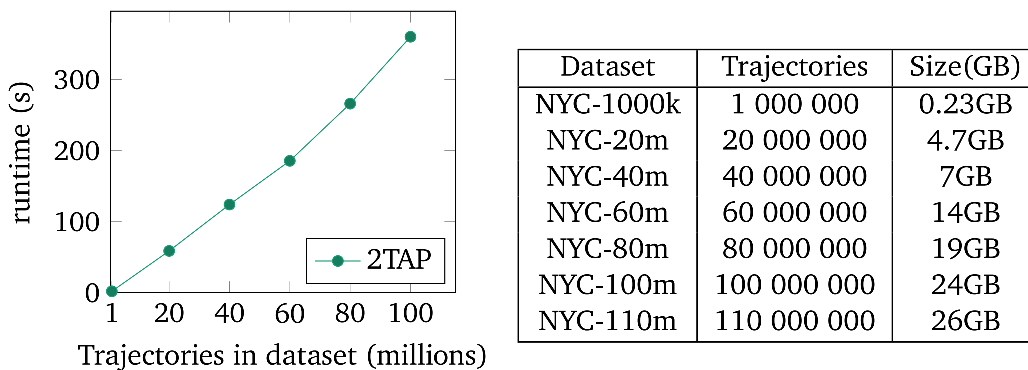


Figure 6.7: Results of the scalability test, comparing 2TA to the parallel 2TAP on a range of large datasets. N = 1000.



Dataset	Trajectories	Size(GB)
NYC-1000k	1 000 000	0.23GB
NYC-20m	20 000 000	4.7GB
NYC-40m	40 000 000	7GB
NYC-60m	60 000 000	14GB
NYC-80m	80 000 000	19GB
NYC-100m	100 000 000	24GB
NYC-110m	110 000 000	26GB

Figure 6.8: Limitation experiment for 2TAP. N = 100.

be limited by the memory capacity of its host-machine.

Finally, we ran a series of tests on 2TAP where we increased the dataset size even further, attempting to push the *dif* machine to its limit. The datasets used are shown alongside the results in figure 6.8. We found that 2TAP on the *dif* machine was capable of processing up to 100 million trajectories. When run on a set of 110 million, all available memory was expended and the application terminated. Despite only occupying 26GB of disk space, the in-memory representation of objects, alongside indexes and the candidate sets amounted to more than the available memory of 384GB.

Dataset	Trajectories	Size(MB)
NYC-10k	10 000	2.7MB
NYC-100k	100 000	24MB
NYC-200k	200 000	48MB
NYC-300k	300 000	72MB
NYC-400k	400 000	95MB
NYC-500k	500 000	119MB
NYC-600k	600 000	143MB
NYC-700k	700 000	167MB
NYC-800k	800 000	190MB
NYC-900k	900 000	214MB

Dataset	Trajectories	Size(MB)
NYC-1000k	1 000 000	238MB
NYC-1100k	1 100 000	261MB
NYC-1200k	1 200 000	286MB
NYC-1300k	1 300 000	309MB
NYC-1400k	1 400 000	333MB
NYC-1500k	1 500 000	357MB
NYC-1600k	1 600 000	380MB
NYC-1700k	1 700 000	405MB
NYC-1800k	1 800 000	428MB
NYC-1900k	1 900 000	452MB

Figure 6.9: NYC synthetic datasets used for scalability testing

6.4 Distributed ETQ

Ellsworth is a result of our endeavor to adapt 2TA to run on clusters. As described in section 5.2, Ellsworth uses the Apache Spark framework and utilizes many of the same principles as 2TA. This set of experiments examines how it scales with data-volume. The experiments run on the *dascosa* cluster, which is detailed in section 6.1.1. Ellsworth is designed with the intention of being able to process datasets orders of magnitude larger than what was tested in the first phase of experimentation. As mentioned in 3.2.2, the number of partitions is an important consideration when tuning Spark in order to maximise resource utilization. Because of this, we run a parameter sweep to find an appropriate number of partitions for each dataset. On the NYC-100m dataset for example, we find that 1337 is appropriate. This is examined further in section 7.3. On the smaller datasets, the number of partitions does not affect the performance of Ellsworth as significantly. Therefore, for the datasets of 1 million trajectories and below, we set the number of partitions to equal the number of HDFS blocks each dataset occupies. NYC-100k only occupies 1 HDFS block whilst NYC-1m occupies 6 blocks, thus the number of partitions is 1 for NYC-100k, and 6 for NYC-1m. We observed in the previous experiments that it_{max} had a large impact on the performance of both 2TA and ILA. Additionally, the optimal it_{max} we found differed greatly from the it_{max} used by Wang *et al.* [5]. Therefore we decided to run a parameter sweep on multiple datasets of different sizes to determine the optimal it_{max}.

In Figure 6.10 we can observe that 1 is by far the best value for it_{max} on the NYC-100m dataset, the same trend is evident for the other datasets as well. We also ran an additional 100 queries at $it_{max} = 100$, which ended up with an average runtime of 188.6 seconds per query. This further reinforces that a low it_{max} is better suited for this environment. This is because Ellsworth, due to the nature of Spark, needs to iterate over every single point to determine if the point is relevant to the current iteration. Additionally, there is a certain overhead to starting new Spark task, by running fewer iterations we reduce this overhead. With the amount of computational power available, the cost of creating and joining the `ScoredTrajectory` instances for each candidate does not outweigh the cost of multiple iterations.

Figure 6.11 shows the average query runtime of Ellsworth compared with 2TAP. The 2TAP results shown in the graph are the same as the ones shown in Figure 6.7. For Ellsworth,

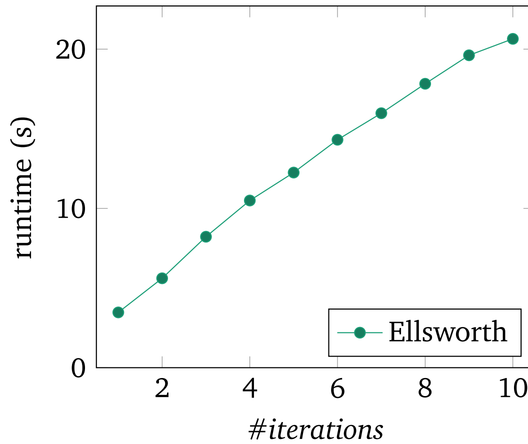


Figure 6.10: Results of it_{max} tests on Ellsworth using the NYC-100m dataset. N = 100.

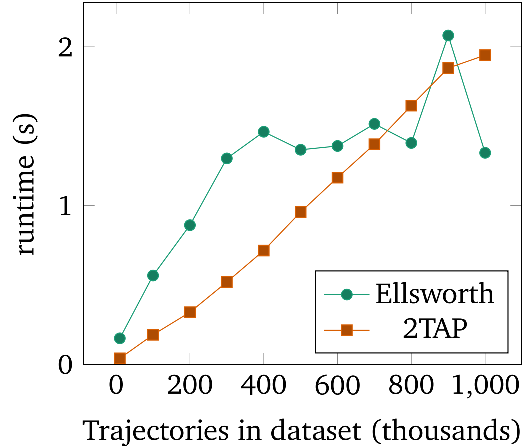


Figure 6.11: Comparing Ellsworth at different to 2TAP on small datasets. N = 100.

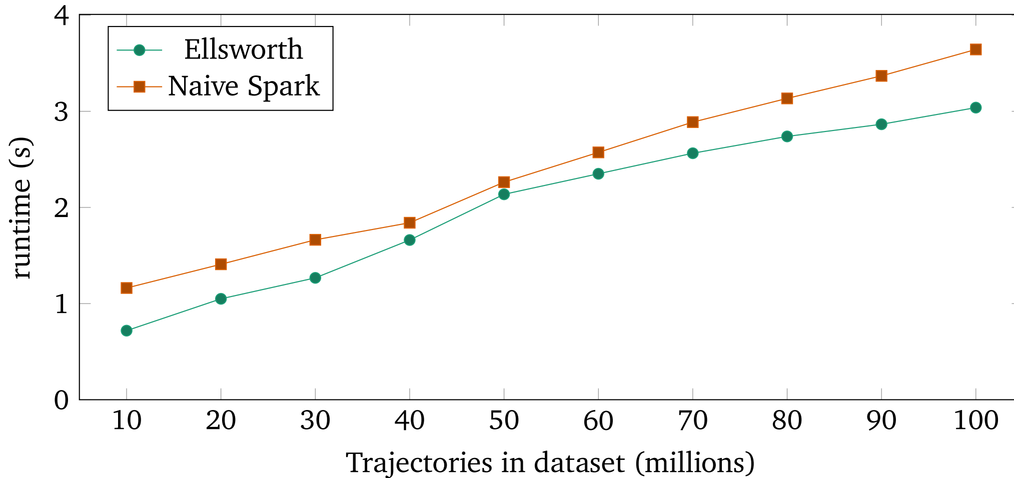


Figure 6.12: Average query runtime of Ellsworth compared to the naive Spark processor

we set the number of partitions to the amount of HDFS blocks each dataset occupies. As we can see, 2TAP is all-around better than Ellsworth on these datasets, which is not unexpected. These datasets are relatively small in the context of HDFS, Apache Spark, and distributed computing as the dataset of 1 million trajectories is only 238MB. The overhead incurred of starting a multitude of new Spark tasks, and the additional network operations required to perform the distributed query, is likely too high to compensate for the additional parallelism gained by using Spark. Additionally, when the runtimes are low, it is hard to discern what time is spent computing- versus runtime caused by other factors, such as variable latency in the network. Therefore, these querying times for Ellsworth at small datasets should be taken with a grain of salt.

Figure 6.12 shows the average query runtime Ellsworth and the naive Spark processor from section 5.3 on significantly larger datasets. The naive ETQ processor and Ellsworth are configured to use the same number of partitions. Here we can see how the increased parallelism and computational power makes the Spark implementation perform orders of

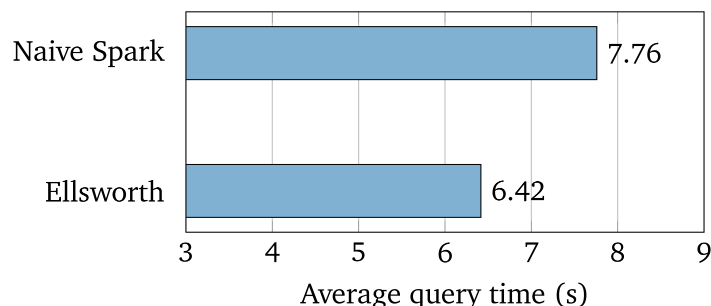


Figure 6.13: Average query runtime of the naive Spark processor and Ellsworth when ran on a dataset of 200 million trajectories. N = 100.

magnitudes better than the results 2TAP shown in Figure 6.8. Both the naive Spark processor and Ellsworth have quite similar performance, although the naive Spark processor has an average query time of 4.16 seconds on the NYC-100m dataset compared to 3.56 seconds for Ellsworth. This suggests that Ellsworth scales a bit better than its naive counterpart. This is somewhat unexpected, as Ellsworth uses a it_{max} of 1, which means that it explores the entire dataset for every query. Since Ellsworth also needs to explore the entire dataset, we expected Ellsworth to be slower than the naive Spark processor, due to the extra steps Ellsworth performs before computing the actual point-wise similarity. However, when it_{max} is equal to 1, the seen upper bound is always equal to the point-wise similarity, so during the *partition local top-k*, Ellsworth only computes the point-wise similarity of $k \cdot partitionCount$ number of trajectories, whilst the naive Spark processor must calculate point-wise similarities for all trajectories. This is discussed further in section 7.3. In order to verify that Ellsworth scales better than the naive Spark processor we ran another experiment, with a dataset of 200 million trajectories. The results are shown in Figure 6.13.

7. Discussion

In this chapter we will discuss the results we observed during the experimentation phase. We will also discuss our other experiences, such as challenges faced while implementing the ILA, 2TA, 2TAP, and Ellsworth algorithms.

7.1 Re-implementing ILA and 2TA

The first phase of experimentation dealt with reproducing the original algorithms by Wang *et al.* [5]. We were able to implement our own *Incremental lookup algorithm* and *Two-Level Threshold Algorithm* that out-perform a naive implementation by a wide margin. However, we encountered a couple of challenges:

- **TkSK**: ILA is very dependent on having an efficient TkSK solution. Our experiments show that a majority (93%) of ILA runtime consists of TkSK queries. A lot of time and effort therefore had to be invested into implementing the RCA algorithm by Zhang *et al.* [8]. The original work did not elaborate on how they implemented RCA, which forced us to make our implementation of RCA based on our understanding of the work.
- **Boundaries and edge-cases**: As mentioned in Chapter 4, an upper-boundary condition for 2TA was either not specified, or incorrectly listed in the original paper. As a result of this, we had to devise a new boundary for use with 2TA.
- **Edge cases**: We also encountered a couple of edge-cases, which were a cause for headache during development.
- **Dataset**: Though several properties of the datasets used are documented, the performance of the algorithms are still dependent on properties that were not. For instance, we do not know what sort of spatial extent the trajectories covered. This is likely to have a high impact on the performance of spatial exploration. Nor do we have any knowledge of the distribution of terms. Without having this sort of data, or even the complete dataset, it is not possible to completely reconstruct the experiments.

We can conclude that both algorithms are indeed far better than a performing ETQ naively, with 2TA outperforming ILA in all cases. We note that the iteration-test results were quite different from the results produced by Wang *et al.*, but this is most likely due to our in-memory approach. As all points are always available through random-access in memory, over-expanding the search are is relatively cheap compared to the time needed for a whole new iteration with several boundary computations.

7.2 Parallelizing 2TA

When performing scalability testing we found that 2TA can definitely handle far larger datasets than a naive approach. ILA performs significantly worse than 2TA. We also found that 2TA can be extended to support basic multi-threading without too much effort. In our experiments we observed that a 2TAP performed about twice as well as 2TA.

We had expected to see more than the 2x improvement over 2TA we observed in the 2TAP tests shown in Figure 6.6, as we ran the experiments on a total of 48 threads, as opposed to a single thread. There are a number of reasons for this, but they all boil down to breaking down task in an efficient manner which can utilize all available resources, and preventing expensive locking when aggregating the subtask-results. We have made note of 5 points which can be improved: (1) The *explore*-phase uses a maximum of $|Q| * 2$ threads, despite the system having more threads available. This leads to unused resources and a loss of potential speedup. (2) Despite launching $|Q|$ `exploreSpatial` calls, each `exploreSpatial` only runs on a single thread. Profiling the code has shown that spatial exploration is far more time-consuming than textual exploration. The `exploreSpatial` iterates over each query points, so breaking this down to several parallelizable sub-tasks task would likely improve performance of this step further, and use more of the available resources. (3) Although we've optimized some locks through using a bucketed system, this only applies to the *expand* stage, not *explore*. Every time a new point is discovered, either spatially or textually, the working thread must aquire and release the lock of the specified bucket. (4) Finally, the whole *explore-and-expand* process itself could be parallelized. If we were to start *expand* while *explore* is still running, we would achieve a higher degree of parallelism and heavily reduce time waiting for straggler threads in between the two phases.

The goal of 2TAP was to apply relatively simple extensions to 2TA, without reworking too much of the original algorithm. If we were to attempt to improve the degree of speedup, the algorithm would have to go through several changes, which was not within the original scope of the task. However, regardless of speedup, we are still ultimately limited by the memory capacity of the host-machine. In our final scalability test we found that 2TAP could handle 100 million trajectories, but crashed when exposed to a dataset of 110 million trajectories due to `OutOfMemory` exceptions.

7.3 Developing Ellsworth

Finally, we investigated if and how distributed computing could improve answering ETQs on large datasets. Our experiments show that Ellsworth scales far better than ILA and 2TA as expected. However it also scales better than the naive Spark processor, which we did not expect at an it_{max} of 1. We expected the naive Spark processor to perform better than Ellsworth, as Ellsworth is forced to explore the entire dataset at this point. When it_{max} is equal to 1 then the grid index only consist of a single cell, and the posting list only consists of one block, which means that every point in the data set is relevant to the first iteration. That means that during the *candidate expansion* step of Ellsworth, every candidate trajectory in every partition will see all the points that belongs to the trajectory. Since there are no unseen points for any trajectory, both the seen upper bound and the seen lower bound of every trajectory is equal to the actual point-wise similarity of the trajectory. The unseen

upper bound of similarity is equal to 0 at maximum iterations, meaning that every single trajectory has a seen lower bound that is higher than the unseen upper bound. This makes the *candidate counting* step of Ellsworth unnecessary; a redundant *count* action that reduces query performance. The unseen upper bound for unprocessed points (Definition 15) is also 0 for all query points, which causes all candidate trajectories to pass the *second-round filtering* step, making it another redundant step. While we iterate over the candidates (sorted by upper bound) in the *partition local top-k* step, only k candidates are iterated over before the actual point-wise similarities of the partition-local top-k trajectories are calculated. In conclusion: when $it_{max} = 1$, Ellsworth is basically the naive ETQ processor with extra steps, which is the reason we expected the naive Spark processor to out-perform Ellsworth. The most likely reason that the naive Spark processor is slower than Ellsworth is that it must compute the point-wise similarity of every single trajectory in the dataset, whilst Ellsworth only has to compute $k \cdot partitionCount$ point-wise similarities. The added overhead of the redundant steps Ellsworth runs through, does not seem to outweigh the cost of computing the point-wise similarities of every trajectory in the dataset. On the NYC-100m dataset, Ellsworth uses 1337 partitions and k was equal to 20 for every query in the experiment. Ellsworth only need to compute $1337 \cdot 20 = 26740$ point-wise similarities while the naive Spark processor has to compute 100 million point-wise similarities. Therefore, we attribute Ellsworth more efficient scaling to the fact that it has to compute orders of magnitude fewer point-wise similarities than the naive Spark processor. In the case of NYC-100m, the naive Spark processor has to calculate 3739 times as many point-wise similarities as Ellsworth.

Choosing the right partition count for each dataset was a big challenge, as there is no clear-cut method to determine the right partition count. We saw huge differences in how fast queries were executed with different amount of partitions, at times up to two or three times slower. The smaller datasets (1 million trajectories and less) were not very sensitive to different partition counts, we therefore settled on used the HDFS block count for these datasets. Running paramter sweeps the bigger datasets (10 million trajectories and more), required a significant amount of time, and would be impractical in real-life scenarios. The results of the parameter sweeps were ambiguous and showed no clear trend compared to the dataset sizes. Therefore we chose partition counts within ranges where the difference in performance was not too large. This means that the number of partitions may not have been strictly optimal, but they should not be drastically worse. Comparing the performance of different Spark implementations is tricky due to the fact that the implementations might have different optimal partition count, and it is difficult to determine how much time is spent on Spark overhead versus actual computation time.

8. Conclusion

In this thesis we have explored the field of spatio-textual trajectory processing, and investigated the work of Wang *et al.* First we reviewed two of their algorithms for performing exemplar trajectory queries, before implementing two of these ourselves: The 2-Level Threshold Algorithm (2TA) and the Incremental Lookup Algorithm (ILA).

Our first research question dealt with whether or not these algorithms were reproducible. During the implementation process we discovered flaws and edge cases, and had to fill in for information that was missing or unclear from the original paper by Wang *et al.* The results we found mostly corresponded to the findings in their source paper, with the exception of it_{max}, where our results diverged due to ours being an in-memory implementation. In order to answer our second research question, regarding the scaling abilities of these algorithms, we made a few extensions to 2TA to enable rudimentary multithreading, referred to as 2TAP. In our experiments we found that the 2TAP algorithm scales well, but is ultimately limited by memory capacity of its host-machine.

Finally, we investigated whether one could achieve better scaling on large sets of data using distributed computing. To accomplish this, we devised a new algorithm based off of the 2TA, named Ellsworth. Ellsworth runs on top of Apache Spark framework for cluster-computing. Ellsworth however, proved to be suitable for processing volumes of spatio-textual data which are orders of magnitude larger than 2TA, let alone ILA. Our final experiments demonstrated that the Ellsworth algorithm was capable of performing ETQs dataset of 100 million trajectories, totaling to over 1 billion points, in about 3.56 seconds.

8.1 Further work

There are many avenues we believe to be worth exploring beyond this thesis. The datasets used for the experiments are all synthesised from the same source dataset, in order to keep a series of properties constant when testing different variables. For example, the $D_{\max}$ value never changes, despite the number of points increasing, leading to a very high point-density. Similarly, the corpus is never expanded regardless of the number of points, meaning the number of total keywords all the synthetic datasets remains the same. If one were to run the experiments on a entire different dataset, which likely exhibits different properties, one may get entirely different results.

Another important note is the fact that our ILA, 2TA, and 2TAP implementations all use data stored in memory. If data were to be stored on disk rather than in memory, we believe the optimal it_{max}-values would look quite different, as this changes the dynamic of exploration time versus computation time drastically. This also severely reduces the prob-

lem where available memory is exhausted when operating on large datasets, at the expense of incurring disk IO.

There is a large variety of indexes that could likely lead to better performance for multiple of the aforementioned algorithms. For instance, we've not experimented using tree-based spatial indexes, such as an R-Tree, or even combined spatio-textual indexes, such as the S2I [3]. On the other hand, we've not put a lot of effort into exploiting the fact that all information is available in-memory. We believe by optimizing based on this fact, one can achieve far better performance on all three of ILA, 2TA and 2TAP.

The issue of indexing and disk-access also applies to *Ellsworth*. If data were to be loaded iteratively in chunks on-demand using some index, rather than processing all points each iteration, one may see more benefit from its iterative aspect on large datasets.

Finally, the process of solving exemplary trajectory queries can be made far more accessible if it were integrated in a larger existing data-processing framework, such as Apache Sedona [25]. Additionally, by having a tighter integration with Spark, one could implement partition-local indexes, which are documented to have a significant performance impact in other spatio-textual spark operations [25].

Bibliography

[1] L. Sloan, J. Morgan, W. Housley, M. Williams, A. Edwards, P. Burnap, and O. Rana, "Knowing the tweeters: Deriving sociologically relevant demographics from Twitter," *Sociological Research Online*, vol. 18, pp. 74–84, Aug. 2013.

[2] L. Sloan and J. Morgan, "Who tweets with their location? understanding the relationship between demographic characteristics and the use of geoservices and geotagging on twitter," *PloS one*, vol. 10, Nov. 2015.

[3] J. B. Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg, "Efficient processing of top-k spatial preference queries," *Proc. VLDB Endow.*, vol. 4, no. 2, pp. 93–104, Nov. 2010.

[4] Y. Yanagisawa, J.-i. Akahani, and T. Satoh, "Shape-based similarity query for trajectory of mobile objects," in *Proceedings of the 4th International Conference on Mobile Data Management*, MDM '03, 2003, pp. 63–77.

[5] S. Wang, Z. Bao, J. Culpepper, T. Sellis, M. Sanderson, and X. Qin, "Answering top-k exemplar trajectory queries," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, Apr. 2017, pp. 597–608.

[6] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval: The Concepts and Technology behind Search*, 2nd. Addison-Wesley Publishing Company, 2011.

[7] A. Gaydhani, V. Doma, S. Kendre, and L. Bhagwat, "Detecting hate speech and offensive language on twitter using machine learning: An N-gram and TFIDF based approach," *ArXiv*, vol. abs/1809.08651, 2018.

[8] D. Zhang, C.-Y. Chan, and K.-L. Tan, "Processing spatial keyword query as a top-k aggregation query," in *Proceedings of the 37th International ACM SIGIR Conference on Research amp; Development in Information Retrieval*, SIGIR '14, 2014, pp. 355–364.

[9] T. Joachims, "A statistical learning learning model of text classification for support vector machines," in *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '01, 2001, pp. 128–136.

[10] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient OLAP operations in spatial data warehouses," in *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, SSTD '01, 2001, pp. 443–459.

[11] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pp. 102–113.

[12] H. Samet, *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., 2005.

[13] Z. Chen, H. T. Shen, X. Zhou, Y. Zheng, and X. Xie, "Searching trajectories by locations: An efficiency study," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010, pp. 255–266.

[14] Y. Zheng and X. Zhou, *Computing with Spatial Trajectories*. Springer Publishing Company, Incorporated, 2011.

[15] M. Kleppmann, *Designing data-intensive applications : the big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, 2017.

[16] A. Hadoop. "Apache Hadoop." (2020), [Online]. Available: `https://hadoop.apache.org` (visited on 05/10/2021).

[17] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, Oct. 2003.

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 2010, p. 10.

[20] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015, pp. 1383–1394.

[21] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013, pp. 423–438.

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012, pp. 15–28.

[23] J. Zhou, N. Bruno, and W. Lin, "Advanced partitioning techniques for massively distributed computation," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 2012, pp. 13–24.

[24] H. Karau and R. Warren, *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*, 1st. O'Reilly Media, Inc., 2017.

[25] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in Apache Spark: The geospark perspective and beyond," *GeoInformatica*, vol. 23, pp. 37–78, Jan. 2019.

A. Edge cases in ILA and 2TA

The pseudo-algorithm for ILA described in Algorithm 1 in section 4.1 and the psuedo-algorithm for 2TA in Algorithm 5 in section 4.2 are slightly modified versions of the original counterparts described in by Wang *et al.* [5]. We made modifications to resolve a number of these edge cases that were not handled, as listed below. All of the edge cases apply to ILA, whilst only edge case 4, 5, and 6 apply to 2TA.

1. On line 8 in ILA Wang *et al.* used the less-than operator instead of the less-than-or-equal operator. The problem with using the less-than operator is that the algorithm will terminate too early, before the TkSK has fetched absolutely all $\lambda^{\max}$ points.

2. Since λ is initialized to k on line 3 in ILA, there may be cases when $\lambda_i^{\max} < k$. In these cases the TkSK will never be run for query point q_i, we therefore added a second clause to the if-statement on line 11 that checks if $c = 0$.

3. On line 12 in ILA Wang *et al.* also used the less-than operator instead of the less-than-or-equal operator. The problem here is similar to the one in the first edge case, it will set $R[c][q_i]$ to $R[c-1][q_i]$ too early, before all $\lambda_i^{\max}$ points have been fetched.

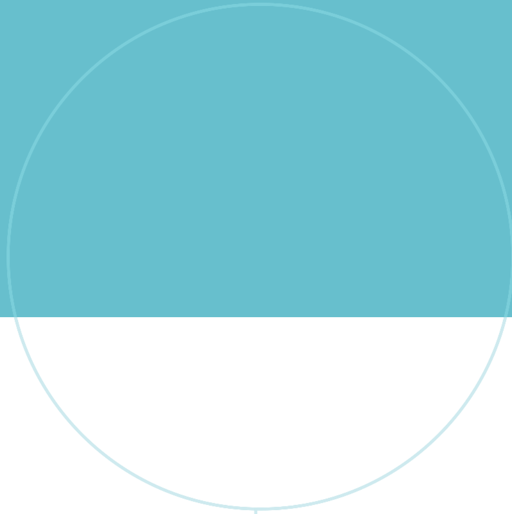
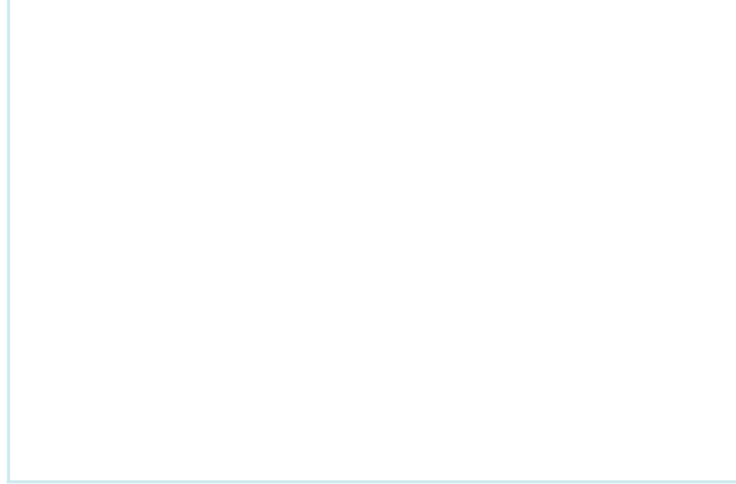
4. On line 20 in ILA and on line 15 in 2TA Wang *et al.* used a greater-than operator instead of greater-than-or-equals operator. The reason it needs to be the latter operator is twofold. Firstly, if $|C_{tra}| = k$ then we still may have enough candidates to satisfy the query. Secondly, there are cases where there only exists exactly k trajectories whose points share at least one keyword with any of the query points, in this case $|C_{tra}|$ will never be greater than k.

5. We changed the greater-than operator that Wang *et al.* had on line 26 in ILA and on line 21 in 2TA to a greater-than operator. If the k-th best lower bound is equal to the unseen upper bound then we can satisfy the query. There may still be unseen trajectories with the exact same similarity, but no trajectories with better similarity. The algorithm should therefore not do another iteration but instead perform the final step and return the result. More importantly, there are cases where we have exactly k candidates and where the k-th candidate's lower bound is exactly equal to the unseen upper bound, if we don't use a greater-than-or-equal operator then the algorithm fails in such cases.

6. We added a break on line 39 that Wang *et al.* did not have. The reason for this is that after the for-loop at line 29 is finished, the algorithm should return the result, because we know that no trajectories can beat the ones in the result. Normally this happens on line 38 when it is discovered that the worst trajectory in the result is better than

the upper bound of the next trajectory. However, in cases where the candidate set contains exactly k candidates the algorithm will never enter the else-statement on line 33. Therefore we added a break so that a new iteration is not started and the algorithm returns its results. This edge case also applied to 2TA since it includes the lines 28 to 39 from ILA.

7. We also have to maintain an array of $|q|$ length for ILA that holds how many points the TkSK has retrieved for each query point q_i. It is initialized to zero and is updated whenever the TkSK retrieves any points. This is necessary because when $\Delta > 1$ then there are cases where $\lambda_i^{\max} \mod (k + \Delta \cdot c) \neq 0$. In these cases λ may jump past the threshold of $\lambda_i^{\max}$ without actually fetching all $\lambda_i^{\max}$ points for query point q_i, which makes the algorithm erroneously re-use the TkSK results from the last iteration whilst there still may up to $\Delta - 1$ points that have not been fetched yet. We therefore added a third condition on line 12 that checks whether all $\lambda_i^{\max}$ have in fact been fetched.

NTNU

Norwegian University of
Science and Technology