NUMA-Aware Spatio-Textual Similarity Join

by

Saransh Gautam

Bachelor of Techonology in Information Technology, Harcourt Butler Technological Institute, Kanpur, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master Of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Suprio Ray, PhD, Computer Science
Examining Board:	Bradford G. Nickerson, PhD, Computer Science
	Rongxing Lu, PhD, Computer Science
	Ian Church, PhD, Geodesy and Geomatics Engineering

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

April, 2020

Abstract

Due to the rapid growth in the usage of location based services, abundant text data is being generated that contain spatial information. Because of this surging combination, spatio-textual queries and in particular, spatio-textual join have gained prominence in recent times. Spatio-textual similarity join is a special kind of operation for finding documents, which are both spatially close and textually relevant. Joins in databases are considered to be the most expensive database operation; similarly spatio-textual similarity join is a highly resource intensive operation. Therefore, it is natural to consider approaches to parallelize this operation.

On the other hand, many modern parallel multi-core systems adopt a NUMA-based memory architecture. NUMA systems entail varying memory access latencies across nodes, which may adversely affect overall query latency. Recent work on spatio-textual similarity join algorithms have not addressed the effects of non-uniform access latencies in multi-node NUMA systems. In this thesis, we explore several approaches to parallelize spatio-textual similarity join on modern hardware. Moreover, we propose three NUMA-aware algorithms. Specifically, we propose a NUMA-aware algorithm that exploits topology-aware work-stealing with adaptive data placement, which performs the best. We evaluate and conduct various experiments with real-world datasets. We demonstrate that our NUMA-aware algorithms perform significantly better than existing approaches that are not NUMA-aware, and in the best case we observe **83**× speedup over the baseline.

Acknowledgement

First of all, I would like to express my deepest gratitude towards my supervisor, Dr. Suprio Ray, without whom this work wouldn't have been possible. He has been a constant source of inspiration and mentor who guided me throughout my journey as a master's student. He always kept his door open whenever I had questions about my research and steered me in the right direction. I have always been influenced by his motivation, immense knowledge and helping nature. I could not have imagined a better supervisor for my thesis.

I am also grateful to my thesis committee members Dr. Brad G. Nickerson, Dr. Rongxing Lu and Dr. Ian Church for their valuable comments and constructive feedback which improved the quality of my work.

I would also like to thank my big data lab mates, Yoann and Puya who aided me in understanding new concepts and for all the discussions we had during our time in lab.

I would also like to acknowledge my supervisor and department of computer science for the financial support throughout the course.

Finally, I express my profound gratitude to my parents and my wife for proving me with unfailing support and continuous encouragement throughout my years of study.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advent of the Global Positioning System (GPS), most devices like mobile phones, car navigation systems and other Internet of Things (IoT) devices are equipped with the ability to locate themselves in near-real time on or near the earth's surface. The last decade has seen an unprecedented rise in social media sites that are generating large amounts of data. This includes various types of data, such as image data, audio, video, and spatial data including the of location of users, sensors and devices along with text-based data describing users, sensors and their activities. In a report on big data, McKinsey Global Institute estimated the volume of spatial data to be about 1 petabyte in 2009 [15] and this is growing at 20% every year. It is estimated that about 80% of data in corporate databases have a spatial component. Spatial data is vital, not just for navigation systems and mapping applications like google maps, but increasingly for analytics in advertising, transportation, energy and related industries. Location specific data along with associated text data offer significant opportunities. Spatio-textual data have been studied and applied for some time now, and a few kinds of spatio-textual queries, such as k-nearest neighbour (kNN) and range queries, have received a lot of attention. An example of range query (circular) is if a person wants to find all the Indian restaurants within 10 km

of her vicinity. Spatio-textual Similarity Join (**STSJ** query) [9] is another kind of spatio-textual query. Given two datasets consisting of spatio-textual documents, an STSJ aims to find all the possible combinations of records from these two datasets that satisfy a spatial distance and a textual similarity criteria. STSJ can be performed on a single dataset as well (self join). As an example, if we want to list all the users of twitter or facebook, who are within 1 km of each other and have 80% match in their tweets (with twitter) or posts (with facebook), we can analyse such cases with spatio-textual join.

A fundamental consideration in spatio-textual similarity join query processing is how to handle the text data. In the domain of natural language processing (NLP), text documents can be treated as string of characters or can be divided into n-grams. In the areas of information retrieval and database query processing, a text document is usually treated as a set of words or tokens. Thus, the text based similarity join problem can be formulated as a set similarity join problem. In recent times, several set similarity join approaches have been proposed, such as ALL-PAIR [11] and PPJ [37]. These algorithms utilize inverted index based approaches with different pruning techniques. Bouros et al [9] proposed Spatio-Textual Similarity Join (STSJ), which integrates spatial pruning and textual pruning to perform spatio-textual join. Star Join [24] integrates an R-tree with ALL-Pair. Most of the approaches are restricted to operate on a single node. A distributed approach by Zhang et al. [38] uses Map-Reduce to make it scalable; we discuss their approach in the related work chapter.

Join is arguably the most expensive of all database operations [8]. Furthermore, queries involving text and spatial data operations are typically more compute intensive compared to those on regular scalar data. STSJ, being a join involving spatial and textual data components, is therefore a complex operation. It is necessary to explore how to efficiently parallelize STSJ queries. Modern multicore machines, commonly based on NUMA, offer ample opportunities for such paral-

Table 1.1: Spatio-Textual Similarity Join Algorithms (baseline and proposed)

Algorithm name	**Description**	**NUMA Aware**	**Partition Method**	**Work Stealing**
SEQ-STSJ-B	**SEQ**uential **STSJ B**aseline	No	Grid	No
NU-STSJ-B	**N**UMA **U**n-aware **STSJ B**aseline	No	Grid	No
NA-STSJ-BP	**N**UMA **A**ware **STSJ B**in **P**acking	Yes	Grid	No
NA-STSJ-Q	**N**UMA **A**ware **STSJ Q**uadtree	Yes	Quadtree	No
NA-STSJ-WS	**N**UMA **A**ware **STSJ** with topology-aware **W**ork **S**tealing	Yes	Quadtree	Yes

lelization. NUMA stands for Non-Uniform Memory Access, which means that the memory system (RAM) is not uniformly distributed across all cores. Instead, the system is organized into multiple nodes, each of which has its own set of cores and memory bank attached to it in a single socket. These nodes are connected to each other via an interconnection network. This gives rise to non-uniform memory access latencies between nodes. For example, if a CPU in node 1 needs to access memory on node 1, it can access that with minimum latency. However, if it is required to access memory of a different node, memory access latency will increase. This is called the NUMA factor [18][25]. Nodes are connected via a high-speed interconnection network, such as Intel's QuickPath Interconnect (QPI) and AMD's HyperTransport (HT). One of the main reasons for the popularity of NUMA machines today is because they provide scalability. Application developers, however, must address variable access latencies by making applications NUMA-aware.

To the best of our knowledge, none of the existing STSJ query algorithms mentioned earlier take NUMA architectures into account. Since joins are computationally expensive operations, reducing the query latency by even a small margin per join operation, can amount to huge time savings within an application. We explore several approaches to parallelize STSJ queries. As with [38] (which is a Map-Reduce implementation of STSJ), we utilize PPJ [37] as the basic algorithm to perform similarity join. In Table 1.1 our algorithms are outlined. SEQ-STSJ-B is a sequential baseline STSJ algorithm, which is used for comparison with our other approaches.

NU-STSJ-B is a parallel version of the sequential algorithm, which is essentially a parallel baseline algorithm designed to operate on a symmetric multiprocessing (SMP) system. The parallel baseline version, NU-STSJ-B, is not NUMA-aware. We develop a NUMA-aware parallel STSJ algorithm, NA-STSJ-BP, by utilizing in-built NUMA library libnuma to support NUMA awareness at the OS level. NA-STSJ-BP uses regular grid based spatial partitioning. Since load balancing is an important factor to consider in any parallel application, NA-STSJ-BP incorporates a bin packing technique to create task partitions, in an effort to distribute the spatial partition evenly among the NUMA nodes. Skewness is a key issue to address in parallel spatial query processing. Therefore, to support skew-resilience, we improve on NA-STSJ-BP and develop a NUMA-aware skew resilient parallel STSJ algorithm called NA-STSJ-Q. It uses a quadtree based spatial partitioning. To further improve parallel performance of STSJ by incorporating dynamic load conditions in load balancing decisions at runtime, we introduce a topology-aware work stealing based technique, NA-STSJ-WS, which utilizes information regarding the topology of the machine. We introduce the idea of remote and local queue per NUMA node. Each node can exploit the information about its nearest neighbours and if it is idle, it can steal work from the remote queue of its nearest neighbours.

We conduct extensive experimental evaluation with several real-world datasets. In our evaluation the performance of NUMA-aware algorithms are compared against our implementation with serial and parallel baseline versions of STSJ. Experimental results show that algorithm NA-STSJ-WS attains about 83× speed-up in STSJ join time over the sequential baseline, and 3.6× speed-up over the parallel baseline.

The contributions of this thesis are as follows:

- To our knowledge, we are the first to consider NUMA-aware parallel spatio-textual similarity join (STSJ).
- We propose three novel NUMA-aware parallel STSJ algorithms: NA-STSJ-BP,

NA-STSJ-Q and NA-STSJ-WS.

- We conduct extensive evaluations with real-world datasets. To establish baselines, we also implement baseline sequential and baseline parallel non-NUMA STSJ algorithms.
- Our proposed algorithms achieve significantly better performance than the non-NUMA baseline algorithms.

1.1 Organization of the thesis

The rest of the thesis is organized as follows. We start with the background in chapter 2 and related work in chapter 3. We then outline the problem in chapter 4 and describe our approaches to a solution in chapter 5. The experimental evaluation details are in chapter 6. Finally, we draw conclusions in chapter 7.

Chapter 2

Background

In this chapter, we provide background related to our work. We discuss background information regarding spatial indexes, space-filling curves, set similarity measures, set similarity join and spatio-textual similarity join. We also discuss about NUMA architecture and NUMA awareness.

2.1 Spatial Indexes

Spatial indexes are data structures used to support efficient access to spatial objects. Such indexes often employ a spatial partitioning technique. R-tree [21], based on minimum bounding rectangle (MBR), is a common data structure for spatial partitioning. Typically, spatial indexes are designed to handle commonly used spatial queries like range query, topk query or finding nearest neighbour. Grid approach is also a very common way to statically divide spaces into spatial grids.

2.1.1 R-tree

R-tree is a tree based data structure for indexing and querying spatial data. It is a variation of B-tree for geometric data. The key idea behind R-tree [21] is the concept of minimum bounding rectangle (MBR). MBR encapsulates one or more than one

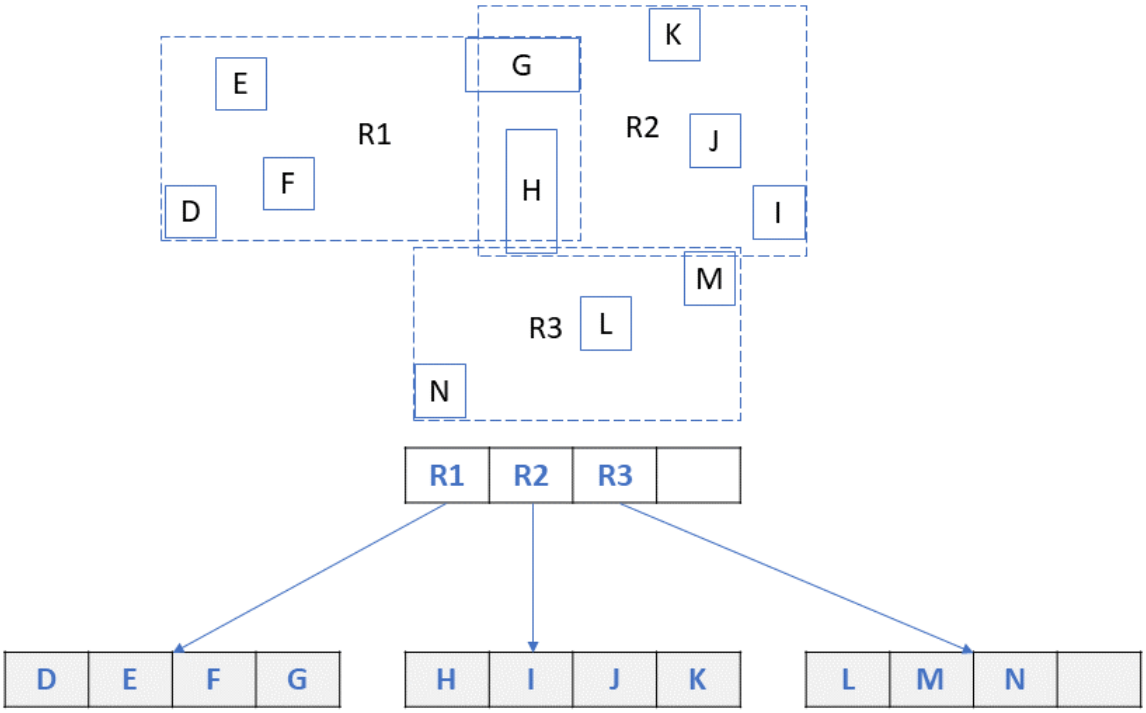


Figure 2.1: An R-tree

spatial objects like point, line or polygon by defining the tightest boundary to cover the object or group of objects. Figure 2.1 shows three MBRs R1, R2 and R3, which contain multiple objects.

R-tree is a height-balanced tree and the objects are stored at the leaf level. In a range search query, the search is performed in a recursive manner where the query object consists of a query bounding box. Starting with the root node, the query box is checked with the intersecting MBR and only the elements in those intersecting MBRs are checked and compared with query object.

There are variants of R-tree such as R*-tree [7] and R$^+$-tree [30], which try to reduce the overlap overhead of R-tree during its construction. They do so by duplicating the object ids over multiple MBRs thereby reducing the query overhead. R-tree can handle multi-dimensional data quite well and are considered the most common data structure for spatial data.

2.1.2 Quadtree

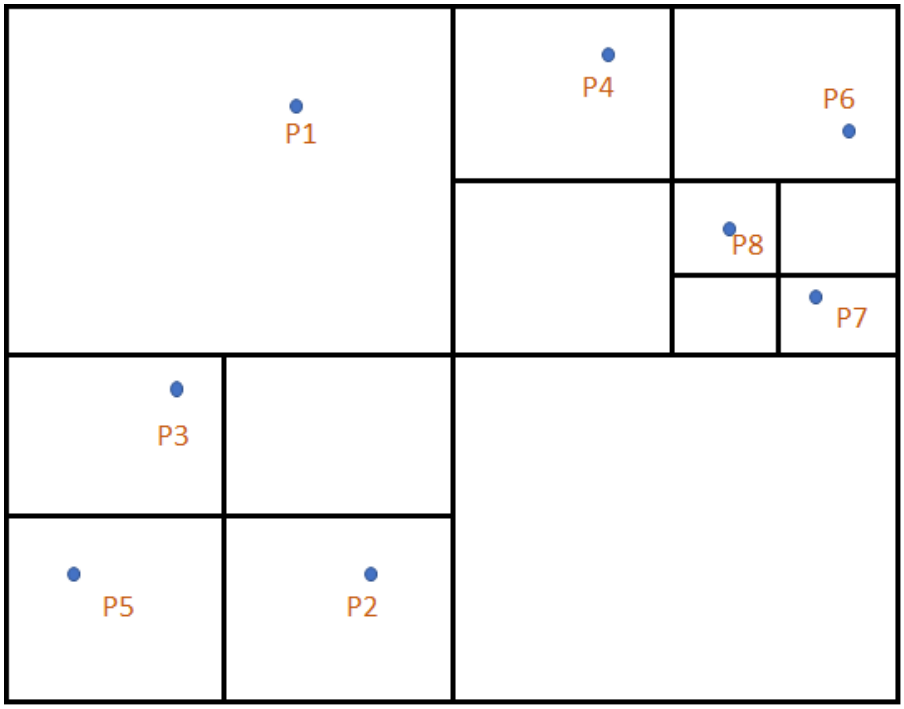


Figure 2.2: A Quadtree representation.

A quadtree [19] is a popular spatial data structure for efficient spatial partitioning and indexing. A quadtree partitions the spatial domain recursively into four quadrants until each partition consists of a certain number of objects, which is termed as the capacity. Figure 2.2 is an illustration of a point quadtree [3]. Due to the strict division restriction of quadtree into four quadrants, a quadtree is an efficient way to partition spatial data, as it can handle skewed data quite well. As Figure 2.2 shows, there are a higher number of partitions in the skewed part of data. Quadtree partitions can follow ordering, such as NW, NE, SW, SE when the spatial objects, such as points, are inserted into it. The range query is fairly simple to support with a quadtree, as the query window can be used to recursively search points from root node in the quadrants in order until it search criteria is satisfied.

2.2 Space-filling curve

Space-filling curve (SFC) [13] is a technique to impose a single dimensional order of traversal for two dimensional objects. Given some two dimensional spatial points, there are various ways to traverse them, for example with grid based partitions, points can be traversed from left to right order. Although it defines a particular order, the traversed points may be away from each other. With SFC, the aim is to visit every point in a 2-dimensional space in such a way that points remain close to each other. Two of the most popular SFCs are Morton's or Z-order curve and Hilbert curve.

2.2.1 Z-order curve

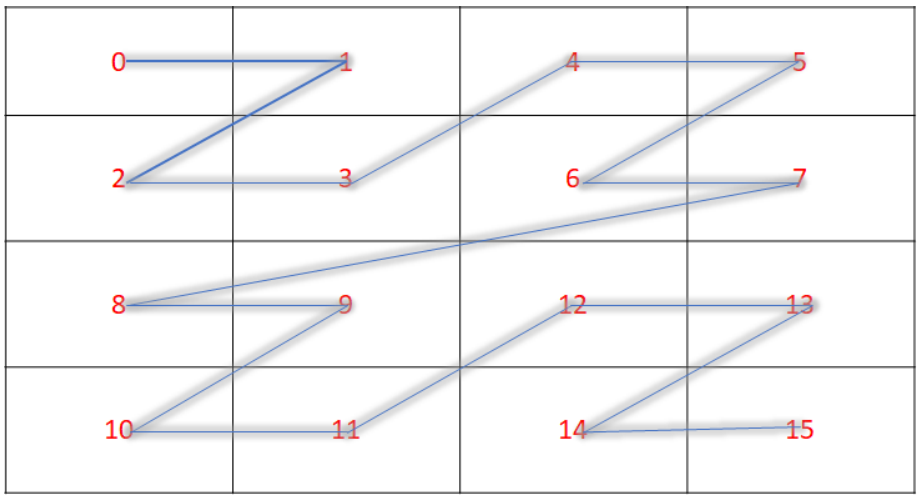


Figure 2.3: Z-order curve.

Z-order or Morton's or Lebesgue curve [5] is locality preserving space-filling curve. Z-order curve is formed by interleaving bits of x and y coordinates for 2-dimensional points. Figure 2.3 shows an illustration of Z-order curve. Quadtree traversal naturally follows Z-ordering.

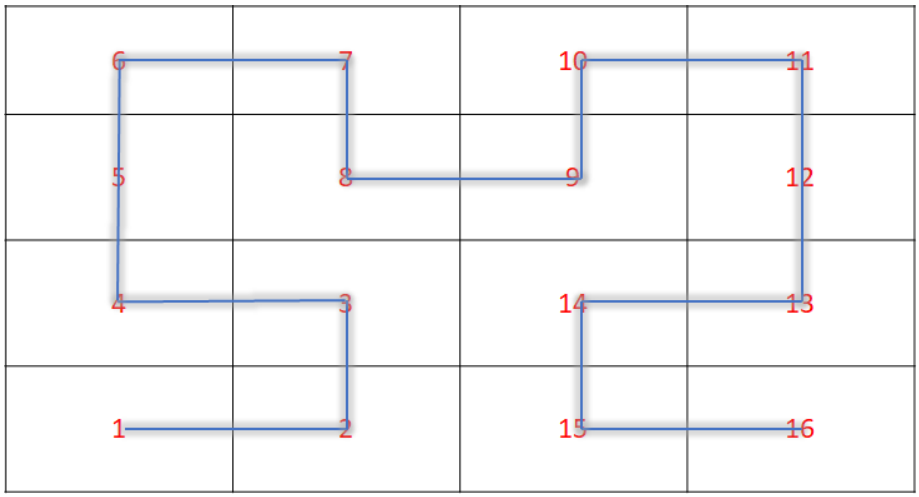


Figure 2.4: Hilbert Curve.

2.2.2 Hilbert curve

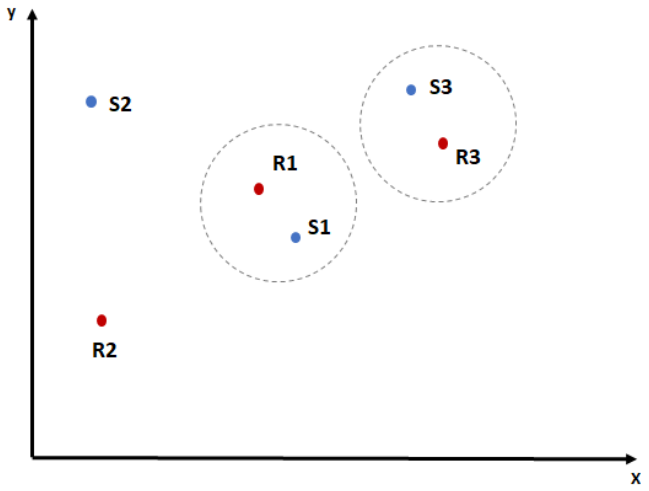
Hilbert curve [1] is another locality preserving space-filling curve, which traverses every point in a $n \times n$ grid where n is a power of 2. It traverses them from top left corner to top right corner following a 'U' type representation. Figure 2.4 represents a Hilbert curve.

2.3 Set Similarity Measures

There are numerous ways to measure similarity between text documents, one of the common ways is to represent text as a set of token of words and compare the tokens based on some set similarity measures. Various set similarity measures exist such as Jaccard's similarity [35] and Sørensen–Dice coefficient [4].

Jaccard's similarity between two sets (of documents) R and S is measured by computing the ratio between the number of common terms and the number of unique terms. Mathematically:

$$J(R, S) = \frac{|R \cap S|}{|R \cup S|} \tag{2.1}$$



R dataset

ID	Text Tokens
R1	Apple, Orange, Banana, Kiwi, Grapes
R2	Kiwi, Papaya, Apple, Mango
R3	Grapes, Coconut, Orange

S dataset

ID	Text Tokens
S1	Banana, Papaya, Watermelon, Orange, Grapes, Kiwi, Pear
S2	Watermelon, Banana, Peach, Plum
S3	Grapes, Coconut, Orange

Figure 2.5: Spatio-textual objects.

As an example, from Figure 2.5 the Jaccard's similarity between $R1$ and $S2$, $J(R1, S2) = \frac{1}{8}$

Similarly, for two sets R and S, Sørensen–Dice coefficient is calculated as:

$$SD(R, S) = \frac{2|R \cap S|}{|R + S|} \tag{2.2}$$

There are, however, other similarity measures to compare text documents. Cosine similarity is a common way to compare documents in a vector space model. A document is represented as a term-frequency vector [22], as is done for term frequency–inverse document frequency (tf-idf) text similarity measures. With this form, the cosine of the angle between two document's textual parts can be computed to give a textual similarity measure $\in [0, 1]$.

2.4 Set Similarity Join

Join in the context of database management system refers to a relational database table join, such as, inner join, left join or outer join. In the context of set similarity join, a join operation aims to find all the combination of pairs from the given set of documents, which satisfy the similarity measure such as Jaccard's similarity.

From Figure 2.5 consider documents $R1$, $S1$. Assume that we are given a similarity measure of Jaccard's with a threshold value of 0.5. In order to be considered $R1$, $S1$ as join pair, both of them must satisfy the similarity threshold, which is as follows:

$J(R1, S1) = 0.5$

Hence, $R1$ and $S1$ will be considered as a join pair. However, if the similarity measure is taken to be more than 0.5, $R1$, $S1$ will not be considered a join pair.

2.5 Spatio-Textual Similarity Join

Spatio-textual similarity join (STSJ) aims to find pairs of documents that are spatially close and textually relevant. Similar to set similarity join, STSJ also looks for join pairs that satisfy a textual similarity threshold based on a similarity measure, such as Jaccard's. However in STSJ, documents are first pruned spatially and the spatially pruned documents then can proceed to the set similarity join phase. As shown in Figure 2.5, document pairs ($R1$,$S1$) and ($R3$,$S3$) are spatially close to each other and can be retained. On the other hand, spatially distant pair ($R2$,$S2$) is pruned before being considered for set similarity join phase. In STSJ documents must satisfy both textual similarity and spatially closeness in order to be counted as join pair, as with ($R1$,$S1$) and ($R3$,$S3$). Also, a join pair is considered between the elements of two datasets R and S. In the case of a self join, documents of a single dataset is joined with itself. Generally, a spatio-textual similarity join consists of five phases as shown below, which we discuss in detail in later chapters:

- **Global Token Ordering** In this phase, all the unique tokens in the documents are listed according to their inverted frequency in a dataset. This gives the less common tokens higher weight compared to the more common tokens. Tokens in each document are then sorted according to global ordering.

- **Index Construction** After computing global token ordering, an inverted index is constructed on the tokens of the build dataset.

- **Spatial Pruning** Typically, documents are first spatially pruned based on a spatial partitioning strategy. This partitioning can be grid based, quadtree based or k-d tree based.

- **Candidate Generation** After spatial pruning, multiple filtering techniques are applied based on set similarity join algorithm, such as, prefix filtering or suffix filtering, in order to generate candidate pairs.

- **Verification** Finally, we generate candidate pairs, and relevant pairs are sent to the verification process where the join is computed.

2.6 NUMA Architecture

With the rising speed of processors in the last few decades, it became imperative for system designers to make sure that the memory access latency in a processor matches the performance improvement of the CPU. However, main memory access latency has not seen such sharp improvement when compared to the speed of processors. Due to this, multi-level caches were introduced to take advantage of spatial and temporal localities.

In shared memory systems, also called Uniform Memory Access (UMA) systems, increasing the number of cores per socket can lead to memory bus congestion and significant performance degradation. Due to these limitations, the concept of NUMA

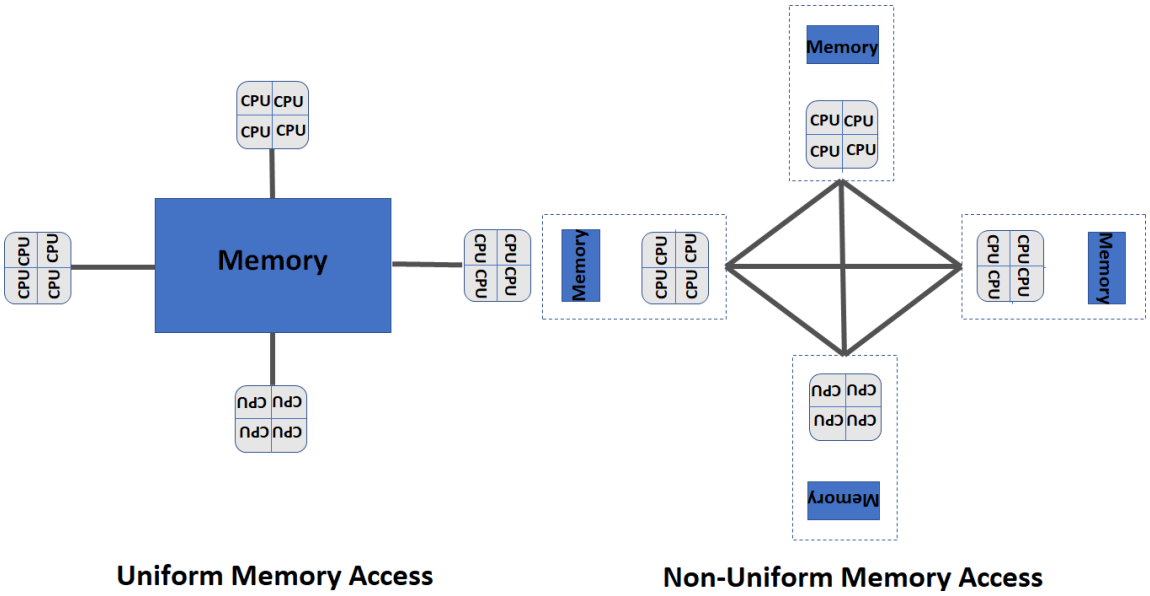


Figure 2.6: UMA vs NUMA Architecture

(Non-Uniform Memory access) was introduced. As shown in Figure 2.6, a NUMA system tries to scale memory access by partitioning the monolithic memory into smaller memory fragments that are closer to a subset of the cores. The cores in a NUMA system are organized into NUMA nodes, where each node consists of multiple cores connected to a single memory bus, with each core having its own caches. A core in each node can access the memory of every other node, but with additional latency to access the memory of remote nodes. Drepper [18] describes the NUMA factors for NUMA machines, which describes the unit overhead for one CPU to access another. For example, with AMD Opetron processor, the NUMA factor is 1.4, which indicates that access to a remote CPU can degrade the performance up to 40%, when compared to local access. As each NUMA node is an independent processing unit with its own memory, another important consideration in NUMA systems is the topology. Topology describes how NUMA nodes are placed with respect to each other. There are some common topologies like star, ring or bus. Each edge between nodes is called a "hop". The number of hops indicates how far the other node is from the local node.

2.7 NUMA-Awareness

Due to the difference in hardware organization of NUMA systems over conventional UMA systems, locality of data plays an important role in determining the performance of any program. Most Operating Systems (OSs) are NUMA-aware, which means, an OS "knows" the NUMA topology of the system and during memory allocation or thread management its decisions are based on topology knowledge. Linux, for example, may migrate a thread to a remote node if it is requesting for a memory region that is on a remote node [16]. Similarly, during memory allocation a Linux kernal uses its built-in APIs, such as kmalloc_node, to allocate memory on the requesting thread's node.

Linux by default follows a **First touch** policy for memory allocation, in which it allocates memory on the same node whose thread first requests it. But there are other available policies, which Linux can follow, if enabled. These can be:

- **Preferred**: The OS will prefer the indicated node to allocate memory on that node. However, if it fails due to any reason, then memory allocation will happen on the closest node.
- **Bind**: Bind applies a strict restriction such that, if memory is not available, its allocation will fail.
- **Interleave**: With interleave, memory is allocated in a round-robin fashion among the nodes, which tries to spread the data across nodes so that the overall effect of NUMA is compensated in the end.

libnuma [2] is a library that provides most of the NUMA related functions a linux kernal also uses. Programmers can use libnuma to make their applications NUMA-aware. There is also a command line interface called numactl, which can be used to run any program with any NUMA policy. Note that numactl defines policies for the

entire program, while libnuma is a library written in C/C++ and its APIs can be invoked from the program. Some of the commonly used functions in libnuma are as follows:

- **numa_alloc_onnode**: It allocates memory on a specific node. It takes size in bytes and node id as parameter to allocate memory. It returns *void**, which holds the address of the allocated node. If memory is not allocated, *null* is returned.

- **numa_max_node**: It returns the maximum node id on a particular NUMA machine.

- **numa_run_on_node**: It runs the task on the core of the node specified. If the data is local to the node, local access happens. Otherwise, the node CPU accesses memory of the remote node.

- **numa_bind**: This function binds the task to a particular NUMA node. After calling this function, all the subsequent memory allocations and process execution happens on the specified node. It take nodemask as a parameter, that corresponds to bit mask of node IDs, containing up to max-node bits.

Chapter 3

Related Work

Join operation in a conventional database system is said to be the most expensive operation. To deal with join, special heuristics are needed due to the massive data computation cost. A spatio-textual similarity join (STSJ) extends the set similarity join to incorporate spatial and textual data. Spatial joins have been an active area of research for a long time. Given a point or line or polygon based dataset, R-tree based techniques have been popular for querying and joining spatial datasets. K-d tree based techniques usually show better performance for point based data, but it degrades considerably as the dimensionality of data increases. In this chapter we look at some of the related works on spatio-textual similarity join and NUMA-aware load balancing.

3.1 Set Similarity Join

Set similarity join algorithms can be categorized into Exact Similarity Joins and Approximate Similarity join. Exact Similarity join methods refer to the join results, which may contain all the possible join pairs that are possible, without containing any false positive or false negative. Generally it requires the data to be pre-computed, thus ensuring every possible case is checked. Set similarity join techniques are used

to compute text based document join. It follows filter and refine steps. One of the first approaches to be proposed for set similarity join was the ALL-PAIR [11] technique, which maintains the inverted index of words in each document. Documents are first sorted according to their inverse frequency in the dataset, positioning the least frequent words first. Documents are considered candidates if they have at least one infrequent word in common. ALL-PAIR [11] uses prefix, length filtering techniques to determine the candidacy of similar sets. ALL-PAIR was further improved by PPJoin [37], which introduced positional filtering to prune candidate sets. Furthermore, PPJoin+ [37] added suffix filtering. Other improvements, such as group join, adapt [33] join and MP-Join [27] extended the set similarity by refining ways to filter out candidate pairs. For instance, group join identifies and form groups based on prefix. Wang et al. [34] improved the existing filter and refine steps of set similarity join by leveraging relations between sets. It improved the filtering step by grouping related sets into index blocks, also skipping the similar set during probe join.

A thorough evaluation by Mann et al. [6] compared seven different set similarity join algorithms with its variations and concluded that among the prefix filtering approaches, ALL-PAIR [11] outperformed the others. Also, on average basic join algorithm PPJ moderately outperformed ALL-PAIR, whereas PPJ+ with suffix filter lagged behind as compared to others as it resulted in significant overhead in their filtering steps.

Approximate similarity join approaches incorporate hashing mechanisms, typically using Locality Sensitive Hashing (LSH) [23] to get all possible nearest neighbors. LSH based techniques like Bayes LSH [29] and Hybrid LSH [39] get the nearest documents in constant time, but results generally contain false positives and false negatives. Approximate join based algorithms are more common in a streaming data systems [31], whereas an estimate of similar documents is sufficient. In this work we

consider only the exact similarity join.

3.2 Spatio-Textual Similarity Join (STSJ)

One of the earliest works on STSJ was star join [24], which is based on a filter and refinement framework and it operates by building spatial and textual signatures. The signature generation phase uses the R-tree based minimum bounding rectangle (MBR) to combine the prefix of the text tokens within the MBR. The idea is to filter out all the documents in a region that does not have any prefix common with the region of other sets. If $\mathcal{M}_p \cap \mathcal{M}_s \neq \phi$, where $\mathcal{M}_p$ and $\mathcal{M}_s$ are the regions in R, such pairs are discarded in the filter step and the remaining pairs are compared using an inverted index during final refinement.

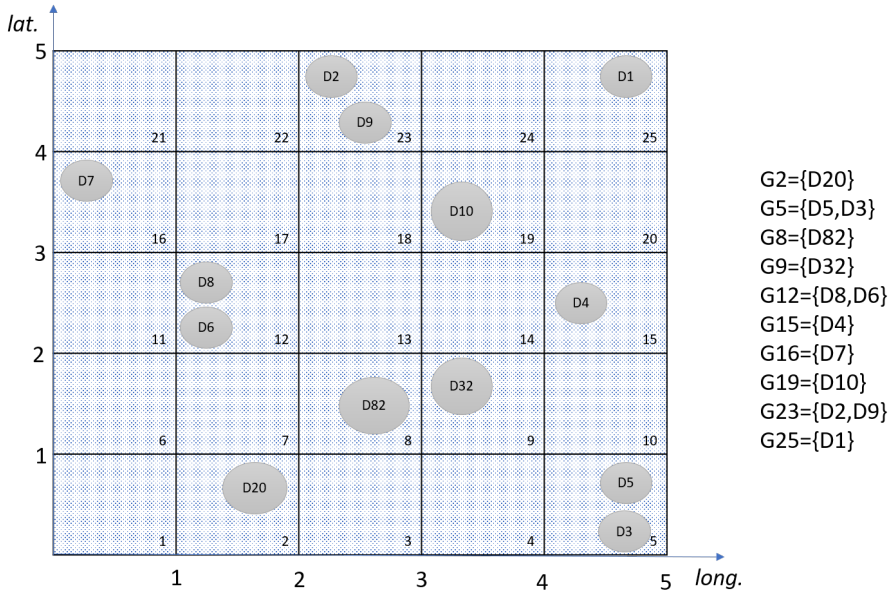


Figure 3.1: Grid Partitioning.

Another STSJ approach uses PPJ and a dynamic grid based partitioning approach [9]. The dynamic grid partitioning method groups the spatial objects based on grid cells such that each cell-width corresponds to the spatial threshold provided. This ensures that objects in any given cell are within a given range. Due to this

partitioning, only the 9 neighbouring cells need to be checked at most to get all the objects of a given cell that are spatially closer to other objects. Further, since the approach uses PPJ as its set similarity join technique, the prefixes of each cell are maintained in an inverted list next to their frequency of uncommon tokens. At each join operation, the cell is only compared with its neighbouring cells and each object's prefix is used to get the object id of similar objects. For example in Figure 3.1, cell 8 is compared with cells 2, 9, 12 as they constitute neighbouring cells, which contain the objects that need to be explored further. This process generates candidate pairs, which are further examined in the verification phase. The authors in [9] improved this baseline by creating per cell inverted index to reduce object join pairs. An R-tree was also used to spatially partition the space as an improvement to the baseline. Xiang et al. [36] proposed block based grid partitioning technique called BIST-JOIN. It uses different heuristics to divide grid space into segments, such as, equal-width segment, equal quantity segment, user defined segment or hybrid segments. These are the variations of grid based partitioning, but they utilize variable grid/segment sizes. Similarly, indexes are also built on the same segments. Some non-grid, tree based techniques include one from Rao et al. [26], which uses quadtree as a partitioning method and introduces two variations of STSJ, based on local and global quadtree partitioning. The local quadtree partitioning approach uses PR-quadtree [28] to divide points into regions and form groups based on spatial threshold. For textual filtering, it uses the ALL-PAIR algorithm. This local quadtree version is also accompanied by a global quadtree version where documents are first partitioned textually using the ALL-PAIR inverted index and then arranged by Z-order, as Z-order is said to be the natural order for a quadtree.

3.3 Scalable Spatio-Textual Similarity Join

Zhang et al. [38] proposed a Map-Reduce based distributed version of STSJ, which made use of grid based partitioning strategy to form independent groups by following grid order. As per our understanding, this is the only distributed, parallel version of STSJ. It usually involved 2 to 3 Map and Reduce phases (depending on the algorithm) to spatially partition data into multiple groups by following a grid order in the Map phase and combined groups in the Reduce phase to compute join based on PPJ+. It also incorporated the cell based index strategy as in [9]. To improve further, it partitioned spatial data using k-d tree by forming bigger groups for Map and smaller groups in Reduce phase by determining the groups in a probabilistic manner.

3.4 NUMA-aware load balancing

Even though OSs, such as Linux, support different NUMA policies, these policies do not always entail the best performance for an application. Due to non-uniform latency, locality is an important factor in NUMA machines. The closer the data is to a processing core, the lower the memory access latency will be. As discussed in [12], even the basic data locality can be more effective than Linux's default first touch policy. Thus, First touch policy of Linux may often adversely affect the performance. Another important factor is load balancing among the nodes. When a new process is created in Linux, the scheduler places the process on runqueue. Also, whenever a child process is created, it is placed in the same runqueue as the parent. In UMA systems, the impact of operating different threads on different cores is not significant. In a NUMA machine, however this is an important consideration, as performance is dependent on the data placement on the NUMA node. This typically becomes more important in the case of load skewness, i.e. if all the threads are scheduled on

the same node. The Linux scheduler uses a load balancing algorithm by exploiting a data structure called shared domain [14], which tries to distribute process load over multiple NUMA nodes, so that no single node has to process all data. Another Linux memory placement policy, Interleave, aims to reduce the contention between the interconnects by interleaving between all the nodes and thus in effect reducing data locality. Since spreading data across nodes reduces contention, for high load applications, traffic congestion can impact performance greatly [20]. The authors in [12] describe a metric based data mapping that tries to take into account both locality and load balancing .

Chapter 4

Problem Statement

In this chapter, we define and formulate our spatio-textual similarity join (STSJ) problem. We outline different parameters and necessary conditions for documents to be considered as join pairs.

We consider a spatio-textual object in the form $o = (id, loc, txt)$, where id corresponds to document id; txt is the set of keywords in the document, $txt = \{w_i : i <= n\}$, n is the number of keywords in text of the document; loc is 2-dimensional geo-location of the document, $loc = (lat, long)$. Then, for any two document sets $\mathcal{R}$ and $\mathcal{S}$, each containing spatio-textual objects, our goal is to find similar pairs of documents, which satisfy the following:

For $r \in R$ and $s \in S$, the spatial distance, $dist(r.loc, s.loc) <= \epsilon$ and textual similarity, $sim(r.txt, s.txt) >= \theta$. Here, ϵ is the spatial threshold and θ is the textual similarity threshold. Note that here ϵ corresponds to the Euclidean distance (L2 norm) and θ is the Jaccard's similarity for any set of tokens txt in document.

Chapter 5

Our Approach

In this chapter, we describe our approaches to parallelize spatio-textual similarity join (STSJ) queries. These approaches are summarized in Table 1.1. First we describe a parallel baseline algorithm NU-STSJ-B, which is not NUMA-aware. Then we introduce the NA-STSJ-BP, our first NUMA-aware parallel STSJ approach that uses grid partitioning. To reduce the effect of skewness, we introduce NA-STSJ-Q algorithm, which tries to form evenly load-balanced groups across nodes by utilizing quadtree partitioning. Finally, we describe NA-STSJ-WS, which incorporates adaptive topology-aware work-stealing and adaptive data placement.

5.1 Preliminaries

Our algorithms utilizes the basic STSJ algorithm discussed in [9]. In this work, Bouros et al. introduced spatial filters on the PPJ [37] algorithm. Although, this algorithm was primarily designed for self join, we made modifications to it to incorporate the join of two datasets. Algorithm 1 is a modified version of the algorithm proposed by Boures et al.

Our algorithm takes datasets $\mathcal{R}$ and $\mathcal{S}$ as input. Where, $\mathcal{R}$ is the build dataset and $\mathcal{S}$ is the probe dataset. This algorithm utilizes ALL-PAIR algorithm, which is

Algorithm 1: PPJ [9]

Input: Given two datasets $\mathcal{R}, \mathcal{S}$ with spatio-textual documents, spatial threshold ϵ, and textual threshold θ

Output: Set $\mathcal{P}$ containing qualified join pairs for $sim(r, s) \geq \theta$ and $dist(r, s) \leq \epsilon$

```
1  Procedure initialize()
2      T ← computeGlobalTokenOrder(R,S) ;
3      Sort R, S by T;
4      I_R ← createInvertedIndex(R,θ) ;
5      P ← ∅ ;
6  Function join(S, I_R)
7      foreach s in S do
8          prefix ← getPrefix(s.txt,Θ);
9          foreach token w ∈ prefix do
10             foreach r in I_R.search(w) do
11                 if dist(r.loc, s.loc) ≤ε & qualifyLengthFilter(r, s)
12                 & qualifyPositionalFilter(r, s) then
13                     if verify(r.txt, s.txt) then
14                         add(r, s) to P
15     return P
16 Function verify(r.txt, s.txt)
17     if JaccardsSimilarity(r.txt, s.txt) ≥ θ then
18         return true
19     return false ;
```

a precursor to the PPJ algorithm and it performs a computation of global token ordering. In procedure *initialize*, a call to *computeGlobalTokenOrder* (Line 2) is used to compute this, which essentially sorts all the tokens from the documents in the dataset in a decreasing order of their tf-idf score. Then each token in $r.txt$ and $s.txt$ is canonicalized by sorting according to global token ordering. As a result, all uncommon words are ordered first, followed by most common word in each $r.txt$. Since $\mathcal{R}$ is the build dataset, we build our inverted index for each token in $r.txt$ (Line 4). Note that the inverted index is built for only the prefix tokens of r.

In the function *join*, we iterate over each document of our probe dataset $\mathcal{S}$. The *getPrefix* function (Line 9) returns the prefixes in each text document. Here, a prefix [37] is given by:

$prefix = |s.txt| - \lceil \theta \cdot |s.txt| \rceil + 1$

For each of these prefix tokens from $s.txt$, the inverted index $\mathcal{I}_R$ is searched (Line 10) for the matching documents in $\mathcal{R}$. Other filtering methods such as length filter and positional filter as per [37] are then applied to the document pair, followed by calculating the distance between the documents. Only the qualified candidate pairs are sent to the verification step where Jaccard's similarity is computed over the pair $(r.txt, s.txt)$ (Line 13). If verified, the pair is added to join set $\mathcal{P}$ and returned.

5.2 Parallel Spatio-Textual Similarity Join Baseline

Due to the compute intensive nature of STSJ queries, it is important to consider how to parallelize it efficiently. As mentioned earlier, in [38] the authors implemented a Map-Reduce-based distributed STSJ, with PPJ-I [9] as the similarity join algorithm. Inspired by it we developed a baseline parallel implementation of STSJ that we call NU-STSJ-B. To parallelize any task, it must be partitioned into sub-tasks such that

each sub-task can operate on a subset of the data. In our approach, we utilize a grid partitioning approach to partition the dataset and traverse them in grid order to form "data groups" or "groups" for short. The group formation steps are the same as in NA-STSJ-BP, discussed in the next section. We use a multi-threading library to parallelize STSJ, such that each thread operates in a different group and let Linux allocate groups to different nodes according to the First touch policy.

5.3 NUMA-aware Parallel Spatio-Textual Similarity Join with Load-balancing

The baseline parallel STSJ approach, NU-STSJ-B, described in the previous section, is not NUMA-aware. In this section we describe our first approach to parallelize STSJ in a NUMA-aware manner. In a NUMA machine each node has its own memory, which is its local memory. An important consideration is how to allocate memory and distribute load evenly among the NUMA nodes. We utilize a bin packing approach to combine the spatial data partitions into groups, so that the workload is roughly evenly distributed among the groups. In the next section we discuss the key ideas behind bin packing.

5.3.1 Bin Packing

The bin packing problem a combinatorial NP Hard problem [10], where there are a finite number of bins and finite items to fit into different bin.
Given a set of bins $b_1, b_2, b_3...b_n$ and a finite number of items with weights $i_1.w_1, i_2.w_2....i_n.w_n$; i represents different items and w are their respective weights, we need to fit the items of different weights into different buckets of certain capacities. Figure 5.1 shows 4 bins with capacity of 10. There are 8 objects with varied sizes, the objects are packed by the first fit bin packing algorithm. There are various algorithms to pack items

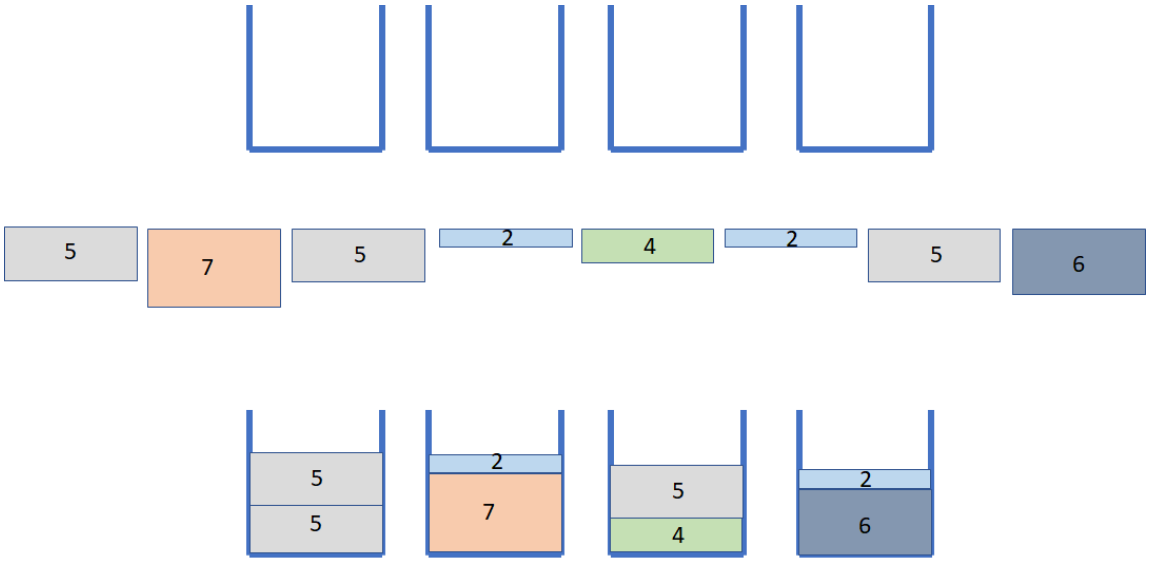


Figure 5.1: Bin packing with bins capacity as 10

into bins [10]. Some of them are:

- **First Fit**: In this approach, the worker puts items in an ordered fashion. It tries to fit the current item into the first bin; if unsuccessful, it goes to the next bin until its finds one.
- **Next Fit**: In this approach, the worker tries to fit the item in the current bin; if unsuccessful it proceeds to the next bin.
- **Best Fit**: Best fit tries to find bins where the item just fits in the bin with minimum difference. Thus, best fit tries to fit items in the least number of bins possible.
- **Worst Fit** : In this case, a worker tries to find a bin with the least number of items and fits the items in that bin.

5.3.2 NUMA-aware Load-balancing

Joins are relatively memory and compute intensive operations, since there are two sets of documents at a time. It is important to consider spatial locality of the doc-

uments so that nearby documents are assigned to the same NUMA node. Further, STSJ query processing involves string comparisons, which are in general quite expensive. Hence, in STSJ, reducing the number of string comparisons can help to reduce the join latency significantly. The ALL-PAIR [11] approach uses an inverted index to ensure that only relevant objects are considered as candidate pairs. In our approach to perform load balancing in NUMA systems, we use a bin packing based approach, in which the length of the documents are used to define the item weight. We first perform a partitioning of the spatial domain using a regular grid. Then we combine the spatial partitions into groups. Since there are a finite number of NUMA nodes, we use bin packing to balance the load among the nodes; in particular, we use a combination of the first fit and next fit algorithms. In our Algorithm (described in Section 5.3.3), we first try to populate all the nodes with our groups in sequential order. This ensures that we have about the same number of groups on each NUMA node.

5.3.3 NA-STSJ-BP Algorithm

The execution flow of our NUMA-aware parallel algorithm NA-STSJ-BP is shown as a flowchart in Figure 5.2. The steps in this flowchart are as follows:

1. **Partition_data** Partition data using spatial declustering into cells and calculate cell sizes.

2. **Form_groups** Form groups with the spatial partitions created in step 1 based on a heuristic (for instance, using a load-balancing approach) and impose an ordering on the groups. Then assign the groups to the NUMA nodes following that ordering.

3. **Create_group_on_NUMA** Perform data placement by allocating memory for each group (formed in step 2) on the assigned NUMA nodes and create index

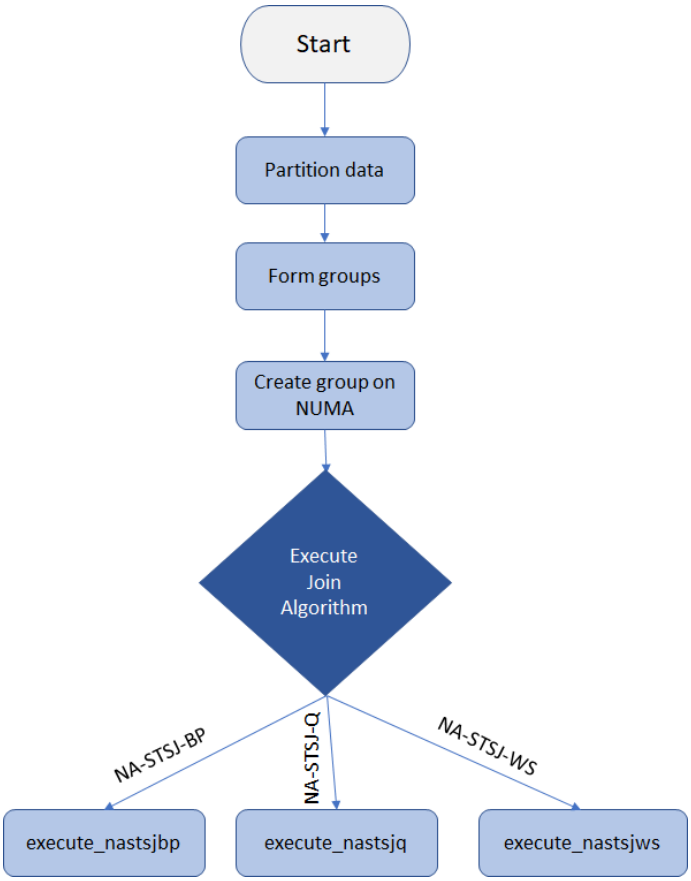


Figure 5.2: Execution flow of our NUMA-aware STSJ algorithms

for each group.

4. **Execute_join_algorithm** Execute join algorithm on each NUMA node in parallel based on the group assignment in step 3. Specifically, we execute the procedure *execute_nastsjbp* for NA-STSJ-BP.

Note that our two other NUMA-aware parallel algorithms (described later) follow similar steps as in the flowchart, however, the details of those steps will differ significantly based on the specific algorithm. Further, depending on the algorithm, a different join procedure will be executed (*execute_nastsjbp*, *execute_nastsjq* or *execute_nastsjws*).

The details of algorithm NA-STSJ-BP is shown in Algorithm 2. We visually illustrate some of the key steps of the algorithm in Figure 5.3. As shown in the figure,

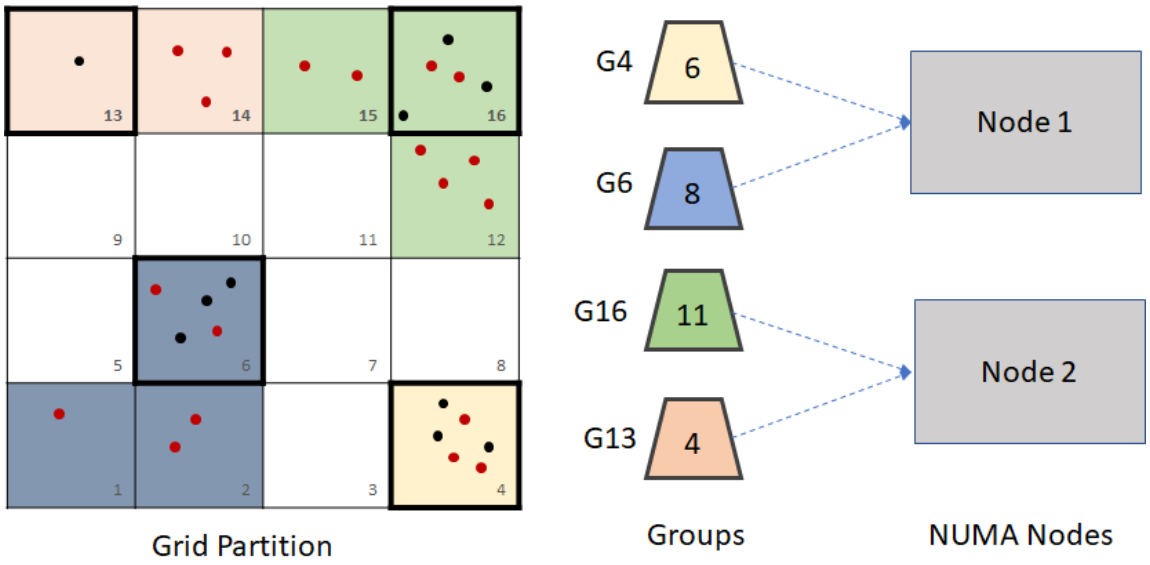


Figure 5.3: Group Formation and allocation on nodes for NUMA-aware Load Balancing

each point represents a document based on its location. Documents in red indicate documents from the build dataset and documents in black are from the probe dataset. The spatial domain is partitioned into grid cells using a regular grid partitioning approach, with each cell width being equal to a spatial threshold. Normally, we choose a spatial threshold that is large enough to cover almost all real-life use cases. This helps in avoiding a reconstruction of the groups if the spatial threshold is varied during query time.

Each cell can contain multiple documents. These grid cells are then utilized to form different groups. In Figure 5.3 four groups are formed: G4, G6, G13, G16. Each group is created around central a cell (highlighted with a thicker border) that contains documents from the probe dataset, along with neighbouring cells that contain documents from the build dataset. The numbers in each group indicate the total number of documents in that group. These groups are then placed on NUMA nodes, for example, G4 and G6 are assigned to Node 1 and G16 and G13 are assigned to Node 2 based on bin packing algorithm (discussed in previous section).

Algorithm 2: NA-STSJ-BP

Input: Given dataset $\mathcal{R}, \mathcal{S}$ containing spatio-textual documents sorted in global order $\mathcal{T}$, spatial threshold ϵ. List of NUMA node workers, $\mathcal{W}$.

Output: Set $\mathcal{P}$ containing qualified join pairs for $sim(r, s) \geq \theta$ and $dist(r, s) \leq \epsilon$

```
1  Grid_R ← createGridPartitioning (R, ε) ;
2  Grid_S ← createGridPartitioning (S, ε) ;
3  C ← ∅ //hash map with cell_id as key and cell_size as value ;
4  nid ← 0 //NUMA nodes id, 0 ≤ nid ≤ max node;
5  N ← ∅ //hash map with cell_id as key and numa_id as value ;
6  G ← ∅ //NUMA groups ;
7  P ← ∅ ;
8  Procedure partition_data(S, ε)
9      foreach s ∈ S do
10         cid ← Grid_S.getGridCellId(s.loc, ε) ;
11         tsize ← getDocumentTextSize(s.txt) ;
12         if C.get(cid) = ∅ then
13             C.put(cid, tsize) ;
14         else
15             C.update(cid, tsize)
16 Procedure form_groups(C)
17     sort C by value ;
18     foreach c ∈ C do
19         gid ← getGroupId(c) ;
20         if G[gid] = ∅ then
21             G[gid] ← initializeNewGroup()
22         if nid ≠ end then
23             N.put(gid, nid) ;
24             nid ← nid + 1 ;
25         else
26             N.put(gid, nid) ;
27             nid ← 0 ;
```

```
27 Procedure create_group_on_numa(𝒩)
28     foreach cell c ∈ Grid_𝒮 do
29         gid ← getGroupId(c) ;
30         nid ← 𝒩.get(gid) ;
31         gcell_R ← Grid_ℛ.getCell(gid) ;
32         𝒢[gid].build.append(gcell_R) ;
33         gcell_S ← Grid_𝒮.getCell(gid) ;
34         𝒢[gid].probe.append(gcell_S) ;
35         ncells_S ← Grid_ℛ.getNeighbourCells(gid) ;
36         𝒢[gid].neighbour.append(ncells_S) ;
37     foreach group g ∈ 𝒢 do
38         nid ← 𝒩.get(g.gid) ;
39         ℐ_g.build ← createInvertedIndex(g.build, θ) ;
40         ℐ_g.neighbour ← createInvertedIndex(g.neighbour, θ);
41         g.Invidx ← ℐ_g ;
42         Allocate g on NUMA node nid using libnuma Bind;
43 Procedure execute_nastsjbp(𝒢)
44     // In each worker node w ∈ 𝒲 do in parallel ;
45     foreach group g ∈ 𝒢 do
46         nid ← 𝒩.get(g.id) ;
47         if nid = w.id then
48             𝒫 ← join(g.probe, g.Invidx); //Algorithm 1
```

Next we describe the details of Algorithm 2 (NA-STSJ-BP). We first take a pass at all the documents in procedure *partition_data* to determine the cell size. This is calculated as the total number of tokens contained in all the documents in a cell. Each cell in $\mathcal{C}$ maintains information regarding cell size. The $form\_groups$ procedure incorporates a combination of first-fit and next-fit based bin packing algorithm, by placing groups in a round robin fashion. We start by sorting our cell map in decreasing order of their cell sizes, which ensures that larger cells are placed in NUMA node first followed by smaller cells. Each group is comprised of a central cell and its neighboring cells. The goal of the group to node assignment is to generate a NUMA node mapping, which links each group of spatial partitions (i.e. grid cells) to a specific NUMA node while following a bin packing algorithm. Line 4 - 5 in Algorithm 2 initializes NUMA node id *nid*, and $\mathcal{N}$ that maps group id to numa id. We start by assigning larger groups to each NUMA node until it reaches last node, *nid* is reset to start node (Line 27) after it reaches the last node. This combination of first fit and next fit algorithm of bin packing allows us to pack groups among each node, with almost equal cell sizes.

After assigning groups to NUMA nodes, we allocate group on specific NUMA node. To do so, we have *create_group_on_numa* procedure. It traverses through every cell in probe grid to create groups and allocates memory for them in the NUMA nodes. Memory placement of group is done by binding current thread to NUMA node id. Build and probe cells are fetched from their respective grids and the rest of the neighbouring cells are appended to create a group. After group creation, we create inverted index for each group. We create an inverted index (Lines 38 - 43) based on both build and its neighbouring cells' documents and the combined inverted index is stored for each group. Data placement is done on particular NUMA nodes for each group. In, *execute_nastsjbp* procedure, we perform the join based on the NUMA groups. We use a threadpool to execute the joins in parallel on different NUMA

nodes. Since each worker thread runs on a specific NUMA node and each group is allocated on a particular node, we execute the join corresponding a particular group on its allocated node $w.id$ (Lines 46 to 49). We utilize the join function from Algorithm 1 and since each group contains its probe documents and inverted index over build dataset, we process join for each group. For a particular group, $g.probe$ represents the probe documents and $g.Invidx$ is the inverted index for that group. Since, data placement for each group is done on a particular node, while traversing groups, we run join for each group by binding them to particular node. We use libnuma *Bind* function, so that only the threads specific to a NUMA node are allowed to run and allocate memory while running on a particular NUMA node. This avoids the usage of the Linux default policy and increases data locality.

5.4 NUMA-aware Parallel Spatio-Textual Similarity Join with Skew-resilience

NUMA-aware load balancing and maintaining locality in NUMA systems are essential, but often depending upon the dataset, skewness also plays an important role in determining overall performance of an application. A real-world dataset can be highly skewed, resulting in some areas to be very dense and other areas sparse. This is especially true with geo-spatial data where large cities can be highly populated, with many geo-spatial features. Due to this skewness, it is important that parallel STSJ approach handle skewness well. Our NA-STSJ-BP algorithm attempts to balance the workload among NUMA nodes to some extent by assigning groups of spatial partitions to NUMA nodes. During actual STSJ execution, depending upon the composition of spatial partitions in each group, significant skewness may be encountered. For example, in dense city areas, it is possible that more similar data is encountered than less populated areas, which will entail significantly more work

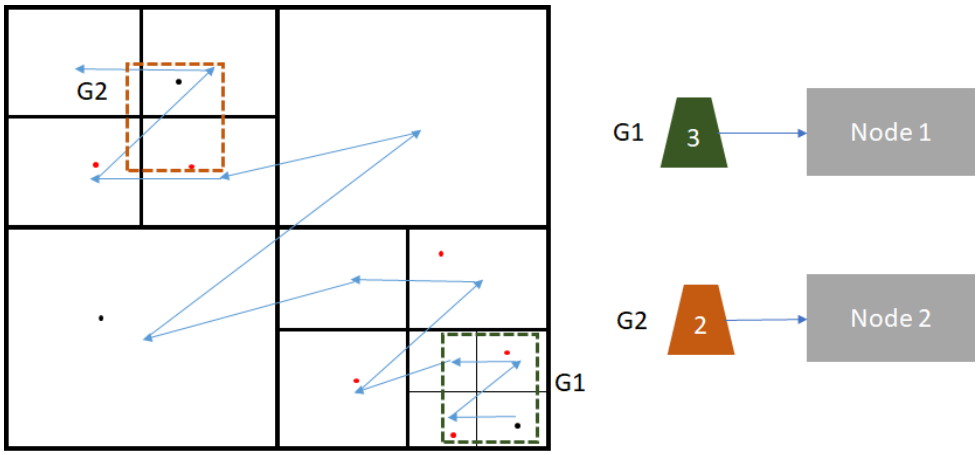


Figure 5.4: Groups assignment to NUMA in NA-STSJ-Q

to the core processing of those dense regions. To address this issue, a spatial data partitioning technique that handles skewness will need to be used. Quadtree based partitioning may be able to handle skewed data better than regular grid partitioning. Whereas grid partitioning defines static grids on the basis of the unit dimension of grid cells, quadtree partitioning recursively subdivides a region into four smaller and equal size regions until the number of objects (i.e. point, line or polygons) in each partition is below than a threshold. Hence, using quadtree partitioning enables us to reduce skewness better.

We utilize quadtree based spatial partitioning to create tiles (spatial partitions). Then we form data groups by imposing Z-order among the tiles and traversing them monotonically according to the Z-order.

We use Z-order traversal in order to preserve locality so that documents which are close to each other spatially, end up in the same node. Furthermore, the data groups are assigned among the NUMA nodes (Figure 5.4) such that they are distributed evenly. These steps minimize the need for duplication of tiles among the node, as documents that are nearby and part of the same data groups are likely to be assigned to the same NUMA node. Note that the groups are formed on the basis of probe dataset (the outer table participating in the join). We select one of the datasets as

probe and one as build.

5.4.1 NA-STSJ-Q Algorithm

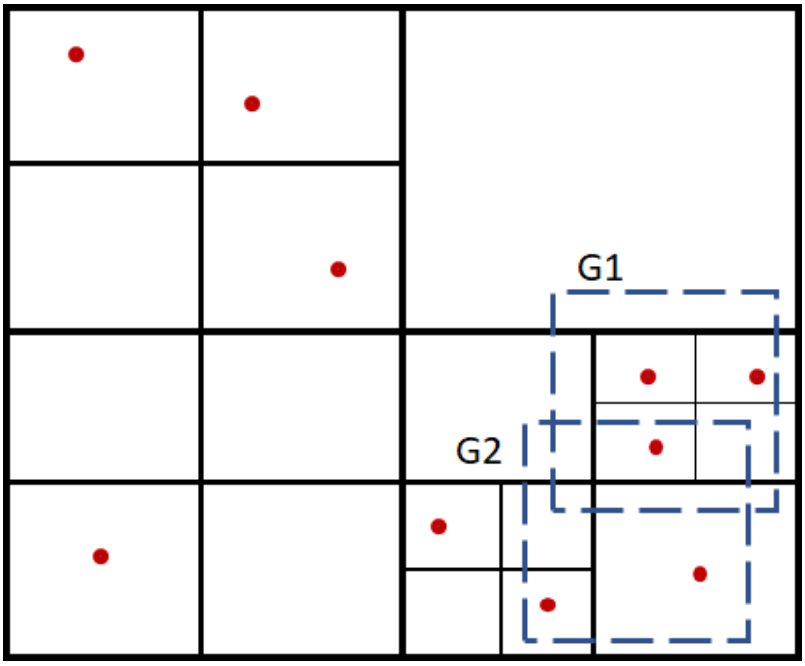


Figure 5.5: Range search in quadtree for group formation

Algorithm 3 describes the steps in NA-STSJ-Q algorithm. We classify our two datasets, $\mathcal{R}$ and $\mathcal{S}$, as build and probe respectively. In this case, the probe dataset is used as the basis of spatial partitioning. Similar to NA-STSJ-BP, NA-STSJ-Q algorithm is comprised of four procedures:

- **partition_data**: this procedure performs the quadtree partition of the documents into cells.
- **form_groups**: this procedure form groups and assigns different groups to different NUMA nodes.
- **create_group_on_numa**: this procedure creates groups and performs data placement on NUMA nodes.

- **execute_nastsjq**: it defines the join procedure of NA-STSJ-Q.

Algorithm 3: NA-STSJ-Q

Input: Given two datasets $\mathcal{R}$ & $\mathcal{S}$ with spatio-textual documents sorted in global order $\mathcal{T}$, List of NUMA node workers, $\mathcal{W}$

Output: Set $\mathcal{P}$ containing qualified join pairs for $sim(r, s) \geq \theta$ and $dist(r, s) \leq \epsilon$

```
1  Q_R ← ∅ //quadtree index for R;
2  ZQ_S ← ∅ ;
3  InvertedIndex ← ∅ // per node inverted index ;
4  nid ← 0 // NUMA nodes id, 0 ≤ nid ≤ max_node ;
5  N ← ∅ //hash map with cell_id as key and numa_id as value ;
6  G ← ∅ //NUMA groups ;
7  P ← ∅ ;
8  Procedure partition_data (R, S)
9      foreach r ∈ R do
10         Q_R.insert(r);
11     foreach s ∈ S do
12         env = Q_R.searchAndGetEnvelope(s) ;
13         idx ← getZOrder(env, s) ;
14         ZQ_s.insert(idx, s) ;
15     sort ZQ_s by key ;
16 Procedure form_groups()
17     partition ← ZQ_s.size()/total_node //total_node is the number of
         NUMA nodes ;
18     lineCount ← 0 ;
19     foreach idx ∈ ZQ_S do
20         if lineCount % partition == 0 then
21             nid = nid + 1 ;
22         gid ← getGroupId(idx) ;
23         if G[gid] = ∅ then
24             G[gid] ← initializeNewGroup()
25         N.put(gid, nid) ;
26         lineCount = lineCount + 1
```

We start by getting both build and probe documents. We first insert documents from $\mathcal{R}$ into a quadtree index (Line 10). Then we iterate over the documents in $\mathcal{S}$ and for each document we obtain the *bounding envelope*, which refers to the bounding box of the leaf level quadtree node that covers the point corresponding to a document

```
27 Procedure create_group_on_numa(𝒩)
28     foreach idx, {s} ∈ZQ_s do
29         {r} ← 𝒬_R.query(idx, ϵ) ;
30         if {r} ≠ ∅ then
31             gid ← getGroupId(idx) ;
32             nid ← 𝒩.get(gid) ;
33             //Create Group, bind node to numa node nid ;
34             𝒢[gid].build.append({r}) ;
35             𝒢[gid].probe.append({s}) ;
36     // create per node inverted index ;
37     foreach worker node w ∈ 𝒲 do
38         InvertedIndex[w.id] ← initialize inverted index ;
39     foreach group g ∈ 𝒢 do
40         nid ← 𝒩.get(g.id) ;
41         InvertedIndex[nid].update(g) ;
42         Enqueue(g) in Join queue 𝒥_nid for node nid ;
43         Allocate g on node nid using libnuma Bind ;
44 Procedure execute_nastsjq()
45     // In each worker node w ∈ 𝒲 do in parallel ;
46     // Run each worker node on numa node nid ;
47     foreach group g ∈ 𝒥_w.id do
48         //Pull groups from Join Queue ;
49         𝒫 ← join(g.probe, InvertedIndex[w.id]) ; //Algorithm 1
```

being searched in the index. We then compute a Z-order code, idx, for each bounding envelope and store them in $\mathcal{ZQ}_s$ (Lines 12 to 14). Here, $\mathcal{ZQ}_s$ is a map used to store the Z-order code (in this case, for the documents in $\mathcal{S}$) as key and a list of references to the documents as value. In the final step of *partition_data*, we sort the $\mathcal{ZQ}_s$ (Line 15) by key.

In procedure $form\_groups$, while traversing Z-order mapping, we partition the entries of index $\mathcal{S}$ equally among NUMA nodes. Also, we get group id and initialize groups, while placing each group in $\mathcal{N}$. $\mathcal{N}$ is a mapping of group id and NUMA node id to which the group is assigned to.

In procedure *create_group_on_numa*, we perform data placement for each group. We start by getting all nearby build documents from $\mathcal{Q}_R$ by doing a range query. Figure 5.5 depicts the group formation process. As shown in the figure, groups G1 and G2 are formed by range query over the quadtree; in reality there will be many more such groups for a STSJ query. The range in range search corresponds to spatial threshold ϵ. We allocate groups on NUMA nodes by getting NUMA node id from $\mathcal{N}$. To allocate groups on NUMA nodes, we use the libnuma *bind* function so that a group is strictly allocated to a particular node.

We create one inverted index per node as part of *create_group_on_numa* function. Then, we initialize inverted index for each node first (Line 38) and while going through each group, we update the index for each node. On each node, groups are maintained in a join queue, which is used during join process. Hence, after creating an inverted index, we insert groups into a Join queue. In the procedure *execute_nastsjq*, we perform our parallel join. Each worker node is launched in parallel with each other. $\mathcal{J}_{w.id}$ represents each node's Join Queue. $w.id$ in this case is equivalent to *nid*, which represents NUMA node id. During the join process, we pull groups from the join queue (allocated previously) and use node's inverted index (index is defined on per node basis in NA-STSJ-Q) to perform PPJ join.

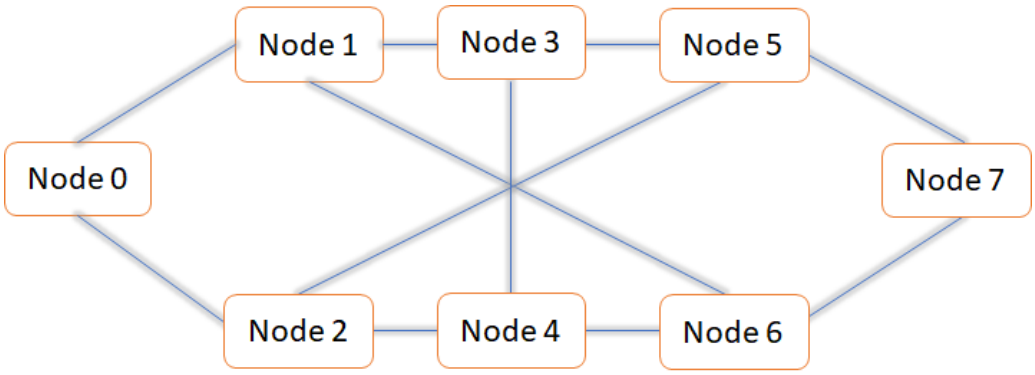


Figure 5.6: Topology of our NUMA Machine.

5.5 NUMA-aware Parallel Spatio-Textual Similarity Join with Topology-aware Work-stealing and Adaptive Data Placement

The main focus of NA-STSJ-Q algorithm is to address data skew and load balancing. Furthermore, with both NU-STSJ-B and NA-STSJ-Q, the assignment of data groups to NUMA nodes happens a priori. Therefore, these two approaches do not take dynamic load conditions into account. Consequently, they do not guarantee that all the NUMA nodes will perform the same amount of work and all nodes will complete processing in the same time. In a real-time situation, some node will finish faster than others, which may depend upon several factors such as, how many string comparisons happen for each node (from the documents), if there are any other OS services or processes running on the node, or how many overall objects comparisons happen. Due to this non-deterministic nature of task execution and completion, it is likely that some node may become idle, while others continue to execute their assigned tasks.

Since a prior assignment of data groups to NUMA nodes do not take into account the dynamic load conditions, the task assignment must be adjustable at runtime. Work stealing is an approach that is known to work well under such circumstances. In the

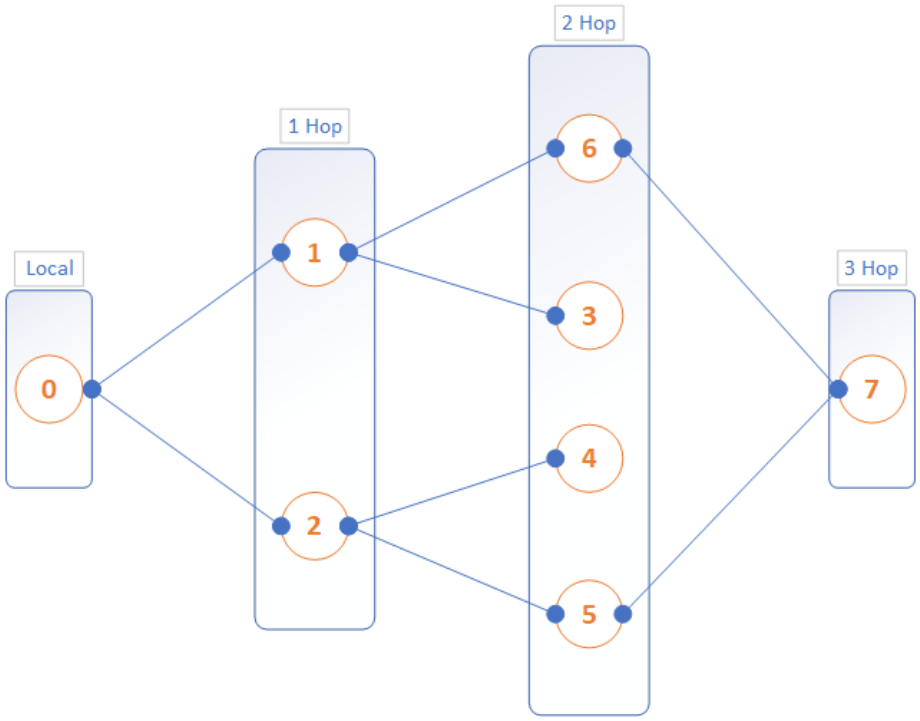


Figure 5.7: Hop diagram with local node as 0.

context of STSJ, work stealing implies that some nodes will be able to steal work from other nodes, while they are working on their join task. However, traditional work stealing is agnostic of the topology of the machine. Hence, due to non-uniform access latencies, the performance of a traditional work stealing based application can deteriorate quickly. If a NUMA node tries to steal work from another node, it depends upon the distance from the local node to the remote node in determining how efficient the work stealing will be. For example, if the remote node is 1 hop away from local node, versus if the local node is 2 hops away from the remote node, it may make significant difference.

For any local node that finishes its own tasks and which tries to steal work from a remote node, it is important that remote access time be minimized. Figure 5.6 shows the topology of our NUMA machine, which we used to conduct our experiments. From Figures 5.7 and 5.8, it can be observed that the hop distance depends on the position of the node. For node 3, most nodes are at most 2 hops away. For node

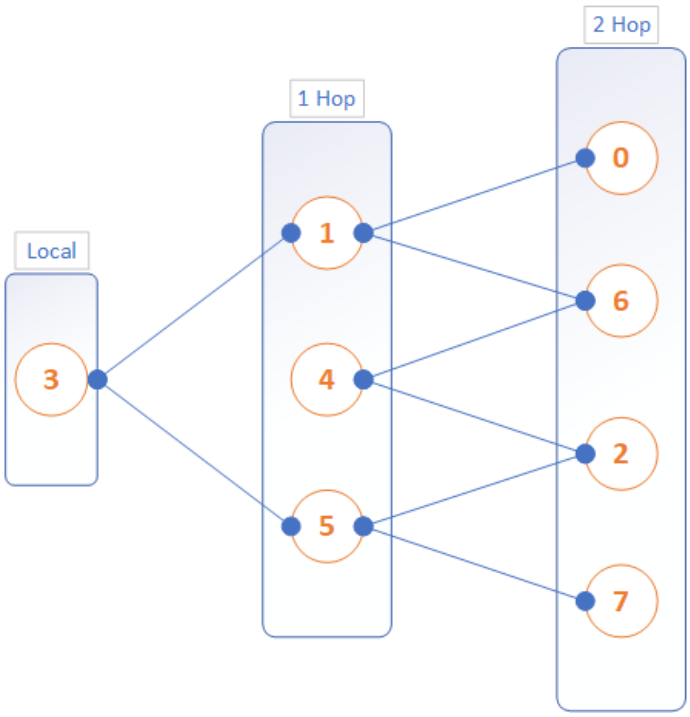


Figure 5.8: Hop diagram with local node as 3.

0 and 7, hop distance can be up to 3 hops for each other. Hence, the distribution of data groups, especially with skewed data, should carefully take into account the 'home' node. In our approach we take into account the topology of the machine while performing work stealing. If a node needs to steal work from a remote node, it first checks its immediate neighbouring node to find if it has any work left to finish and if so, work can be stolen from that node. If not, it jumps to the next nearby node. To support this, we introduce the concept of local and remote queues, which is described in the next section. After completing local tasks, a node moves on to the next closest node that has any work left in its remote queue. This search continues until it finds the node, which is still working.

5.5.1 Topology-aware Work Stealing

Topology-aware work stealing becomes necessary for any application if it is running on a NUMA machine. Therefore, we need some mechanisms to support such work

stealing. We define two queues per node. One is a *local queue* and the other is a *remote queue*. After we segregate documents into data groups based on their locations and assign the data groups to different nodes, we need to process each group to run join for their similar documents in both the build and probe datasets. Hence, the formed groups are either put in the local queue or the remote queue. Tasks from the local queue are solely processed by the local node and any task in the remote queue can be processed by remote nodes. Each node has to process tasks in its own local and remote queue first before it can steal from other nodes' remote queues. The ratio of the number of task elements (i.e. data groups) that are on the local queue to that of remote queue is called the **dynamic ratio** and is defined as follows:

$$\mathcal{D}_r = \frac{\mathcal{R}}{\mathcal{R} + \mathcal{L}} \tag{5.1}$$

where $\mathcal{L}$ is the number of data groups in the local queue and $\mathcal{R}$ is the number of data groups in the remote queue. Also, total number of data groups ($\mathcal{T}$) in a node is as follows:

$$\mathcal{T} = \mathcal{R} + \mathcal{L} \tag{5.2}$$

Dynamic ratio enables us to adjust the access to remote nodes' tasks (data groups) based on the skewness of the data. It should be noted that remote access will increase the memory access time to remote queue based on NUMA factor (discussed in Section 2.6). The dynamic ratio can vary between 0 to 1, where 0 indicates that all the data groups are placed in the local queue, whereas 1 means that all the groups are in the remote queue. We found that for our dataset 0.5 was the optimal dynamic ratio. It can vary depending upon the number of nodes in a particular NUMA machine and also the properties of dataset.

5.5.2 NA-STSJ-WS Algorithm

NA-STSJ-WS utilizes a topology-aware work stealing to perform parallel NUMA-aware STSJ. NA-STSJ-WS modifies the existing NA-STSJ-Q and introduces work stealing during the join process. We use the *partition_data* and $form\_group$ from Algorithm 3. In the *create_group_on_numa* procedure, after creating groups and allocating them on respective nodes, we create local and remote queue as discussed in 5.5.1. We replace the join queue from NA-STSJ-Q Algorithm 3 (Line 42) with local and remote queues. Then groups are assigned to local and remote queues based on the dynamic ratio Algorithm 4 (Lines 10 - 16).

The *execute_nastsj* procedure in Algorithm 4 describes join process during runtime. We take NUMA workers as input, where each NUMA worker is associated with groups and inverted index of build documents with its remote and local queue (in *create_group_on_numa*). NUMA node map $\mathcal{N}_{map}$ enlists each node with its neighbouring node in order of hops (see Figure 5.7). *execute_nastsjws* launches the join process on each node in parallel. As discussed in section 5.5.1, each node tries to process its local queue first and then moves on to its own remote queue. Lines 19 -26 corresponds to the local join process. In this, first current node gets groups from its local queue and proceeds to call PPJ join algorithm. After processing all groups from the local queue, a node's remote queue is peeked to see if any group is present in the queue. Note, since it is remote queue, other nodes can steal groups from it. Once, a node is finished with its local and remote queue (Line 26), its proceeds on to work-stealing from the remote queues of its nearest neighbors and performing join (Lines 27 - 36). Current node first gets the nearest neighbor node from the topology map $\mathcal{N}_{map}$. It peeks at the remote queue of the nearest neighbor node (Line 32), and if any unprocessed group is present, its starts by removing the group from the back of the queue (instead of from the front of the queue) and runs the PPJ join algorithm on the removed (i.e. stolen) group.

Algorithm 4: NA-STSJ-WS

Input: Given two datasets $\mathcal{R}$ & $\mathcal{S}$ with spatio-textual documents sorted in global order $\mathcal{T}$. List of NUMA node workers, $\mathcal{W}$, node map for node topology detail, $\mathcal{N}_{map}$, running_list is list of workers processing their respective nodes, $\mathcal{J}_{nid}$ is the node level join queue

Output: Set $\mathcal{P}$ containing qualified join pairs for $sim(r,s) \geq \theta$ and $dist(r,s) \leq \epsilon$

1 // Global variable initialization steps are same as in Algorithms 3
2 **Procedure** `partition_data` *($\mathcal{R}$, $\mathcal{S}$)*
3 //All steps are same as in Algorithm 3
4 **Procedure** `form_groups()`
5 //All steps are same as in Algorithm 3
6 **Procedure** `create_group_on_numa(`$\mathcal{N}$`)`
7 //Same steps as lines 29 to 43 in Algorithm 3
8 **foreach** w *in* $\mathcal{W}$ **do**
9 $running\_list.add(w)$;
10 $LQ_{w.id} \leftarrow \varnothing$ //Local Queue ;
11 $RQ_{w.id} \leftarrow \varnothing$ //Remote Queue ;
12 **foreach** *group* $g \in J_{w.id}$ **do**
13 **if** $LQ_{w.id}.size \geq (\mathcal{D}_r * \mathcal{J}_{w.id}.size)$ **then**
14 $LQ_{w.id}.insert(g)$
15 **else**
16 $RQ_{w.id}.insert(g)$
17 **Procedure** `execute_nastsjws(`$\mathcal{W}$`)`
18 // In each worker node $w \in \mathcal{W}$ do in parallel ;
19 // First look at own Local queue ;
20 **while** $LQ_{w.id}.peek() \neq \varnothing$ **do**
21 $g \leftarrow LQ_{w.id}.remove()$;
22 Call Algorithm 1 with $join(g.probe, InvertedIndex[w.id])$;
23 // Then look at own Remote queue ;
24 **while** $RQ_{w.id}.peek() \neq \varnothing$ **do**
25 $g \leftarrow RQ_{w.id}.remove()$;
26 Call Algorithm 1 with $join(g.probe, InvertedIndex[w.id])$;
27 // Work-stealing: then look at nearest node's Remote queue ;
28 **while** *true* **do**
29 $nearest \leftarrow \mathcal{N}_{map}.next\_nearest(w, nearest)$;
30 **if** $nearest \neq \varnothing$ **then**
31 $nrid \leftarrow nearest.id$;
32 **if** $RQ_{nrid}.peek() \neq \varnothing$ **then**
33 $g \leftarrow RQ_{nrid}.remove\_from\_back()$;
34 $\mathcal{P} \leftarrow join(g.probe, InvertedIndex[nrid])$; //Algorithm 1
35 **else**
36 **break** ;

To join groups from local and remote queues of a local node, we use the libnuma *bind* function. Whereas, for remote join on remote node, we use libnuma *run_on_node* function.

Chapter 6

Experimental Evaluation

In this chapter, we describe our experimental evaluation that we conducted with real-world datasets. We present performance evaluation of our NUMA-aware algorithms by comparing them with non-NUMA baseline STSJ algorithms that we also implemented.

6.1 Experimental Setup

Experiments were conducted on a machine with dual-core AMD Opteron processors having a total of 16 cores. These cores were equally divided among 8 NUMA nodes. Each node contains 16 GB of memory, with an aggregate memory of 128 GB. The architecture of our NUMA machine is shown in Figure 5.6.

6.2 Datasets

Four datasets are used in our experiments. Three of them are Twitter [32] datasets with different sizes of 200k, 2 million, 20 million. Twitter datasets contain tweets with their geo-graphical locations. Each Twitter dataset contains on an average 5 keywords per tweet. Another dataset, Spaten [17] has about 1.5 million restaurant reviews across the US. The Spaten dataset also contains geo-location with each

Table 6.1: Datasets Details

Dataset name	**Num. of documents**	**Average keywords**	**Max keywords**	**Average doc. length**
Spaten	1,523,849	21.75	62	23.26
Tweets200k	200,000	5.08	30	25.58
Tweets2mi	2,000,000	5.06	70	25.55
Tweets20mi	20,000,000	5.48	38	5.70

Table 6.2: Experiment settings

Parameter	**Settings**
Datasets	Tweets200k, **Tweets2mi**, **Spaten**, Tweets20mi
Spatial Threshold	**0.5**, 1
Textual Threshold	**0.8**, 1
Dynamic Ratio	0.2, **0.5**, 0.7, 0.9
Number of threads	1, 2, 4, 8, **16**

review. It contains an average of 21 keyword tokens. Stop-words from their text keywords were removed from each of the datasets. Table 6.1 shows more detail about the datasets.

6.3 Evaluation

We conduct our experiments based on various parameters. We vary algorithms, number of threads, dynamic ratio, and also build time. Table 6.2 lists the important parameters for our experiments. We use our NU-STSJ-B, NA-STSJ-BP, NA-STSJ-WS algorithms for evaluation. Since NA-STSJ-WS utilizes NA-STSJ-Q and the initial phases for both algorithms are the same, we only report NA-STSJ-WS for performance evaluation. We chose Tweets2mi and Spaten as our default datasets. Tweets2mi is the default build dataset and Spaten is the default probe dataset. Default values are shown in bold font in Table 6.2. In the following sections, we discuss the evaluation results in detail.

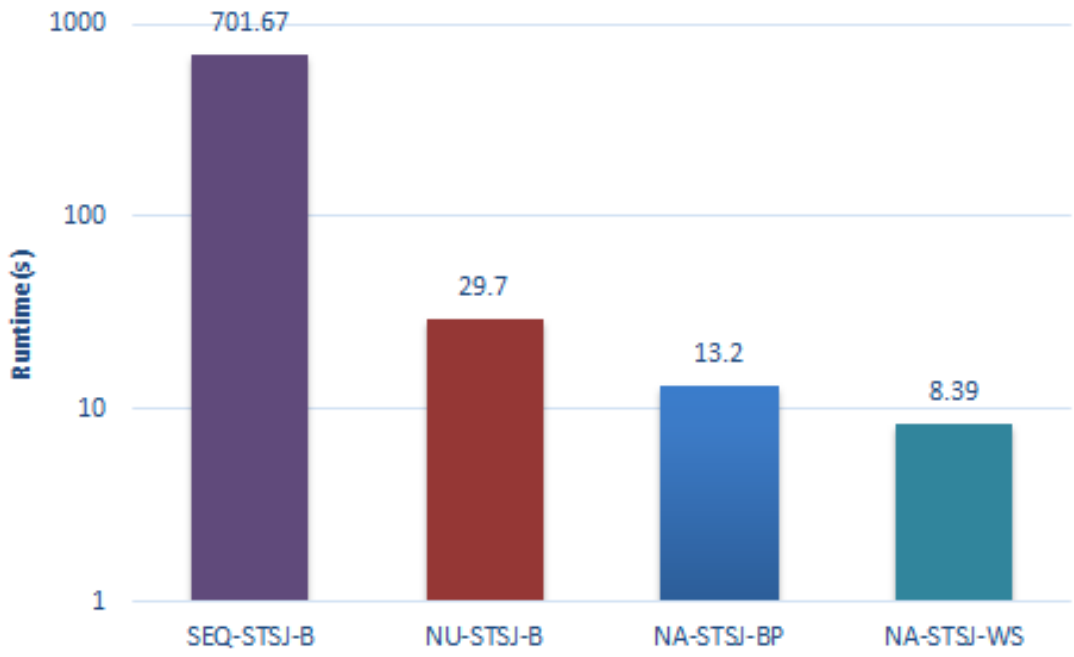


Figure 6.1: Vary Algorithms

6.3.1 Vary Algorithms

We vary our different algorithms and we report the STSJ query execution time. In all of these algorithms PPJ is used as the basic set-similarity join algorithm. Our sequential baseline algorithm is SEQ-STSJ-B, which is a single threaded baseline version, whereas NU-STSJ-B is a parallel baseline algorithm. None of these two algorithms are NUMA-aware. Against them we compare our NUMA-aware parallel algorithms NA-STSJ-BP and NA-STSJ-WS. In Figure 6.1 we report the speedup achieved by the parallel algorithms over the sequential baseline version. As shown, all algorithms achieved considerable speedup over the sequential baseline algorithm. Our NA-STSJ-WS shows about about **83**× speedup, when compared to the sequential baseline version of STSJ. Whereas, NU-STSJ-B and NA-STSJ-BP show 24× and 53× speedup, respectively. When we compare our three parallel NUMA-aware algorithms against the non-NUMA parallel baseline NU-STSJ-B, they generally perform better than the baseline and NA-STSJ-WS achieves the best speedup. For instance,

NA-STSJ-WS and NA-STSJ-BP achieved speedups of **3.6**× and 2.25× respectively.

6.3.2 Vary Datasets

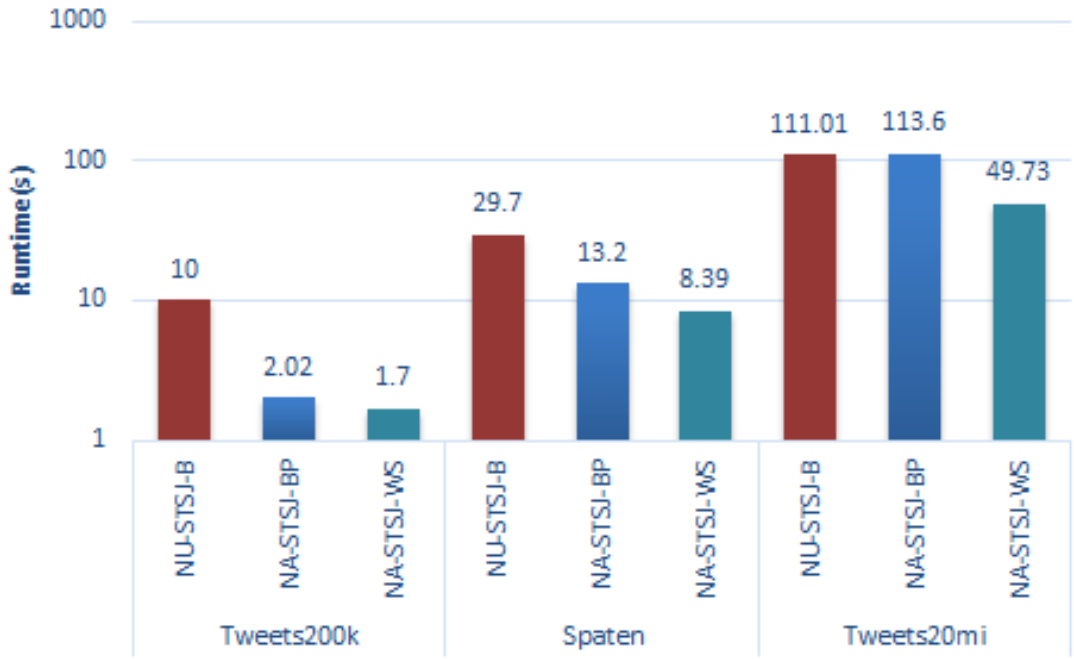


Figure 6.2: Vary Datasets.

Figure 6.2 depicts the runtimes of different algorithms by varying different datasets. We vary Tweets200k, Spaten, Tweets20mi datasets as probe datsets and keep the Tweets2mi fixed as the build dataset. As figure 6.2 shows, NA-STSJ-WS outperforms others for all the datasets. Even for the biggest dataset Tweets20mi, NA-STSJ-WS was more than twice as fast as the other two algorithms. This shows the effectiveness of topology-aware work stealing, particularly, in terms of handling skewed data. The other NUMA-aware approach, NA-STSJ-BP statically distribute the work-load among the NUMA nodes to support load balancing, whereas NA-STSJ-WS can take into account dynamic load conditions. Since the longest distance between two NUMA nodes is 3 hops in the NUMA system we used, stealing work from remote nodes can be expensive, particularly for large dataset. Hence, there is a trade-off

between static vs dynamic workload assignment. Our topology-aware work-stealing does a reasonably good job in this regard. Understandably, the effectiveness of NA-STSJ-WS is somewhat reduced as dataset size increases.

6.3.3 Vary Threads

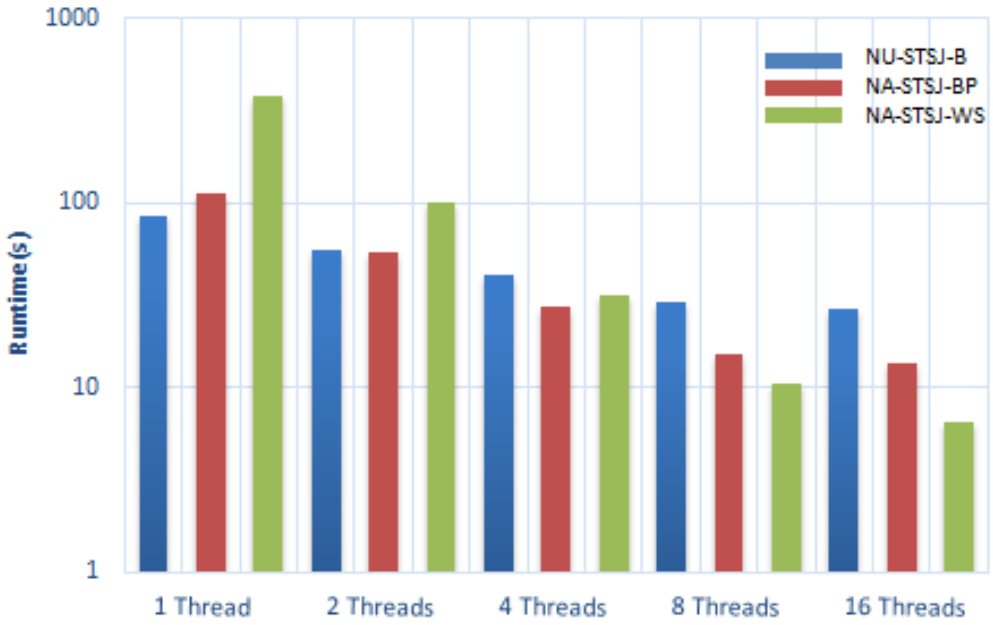


Figure 6.3: Vary number of threads

We vary number the number of threads as we execute each algorithm. As shown in figure 6.3, NA-STSJ-WS performance improves as we increase number of threads. Noticeably, the single-threaded performance of NU-STSJ-B is better than the others; this was expected as a relatively larger number of groups are formed in NA-STSJ-WS as compared to other two algorithms. However, as the number of threads are increased, the performance of NU-STSJ-B does not improve much, in contrast to that of NA-STSJ-BP and NA-STSJ-WS. The performance of NA-STSJ-WS drastically improves with an increase in the number of threads. This is mainly due to more efficient locality of groups and work stealing. Also, NA-STSJ-BP shows considerable improvement due to load-balancing (with our bin packing based approach), as the

groups get evenly distributed among the NUMA nodes. During execution, improved locality helps NA-STSJ-BP to gain significant speedup even when NA-STSJ-BP and have same number of groups. Figure 6.4 depicts the effects of number of threads on NA-STSJ-WS, as our NUMA system have 16 hardware threads (2 per NUMA node), the runtime of NA-STSJ-WS is minimum for 16 threads.

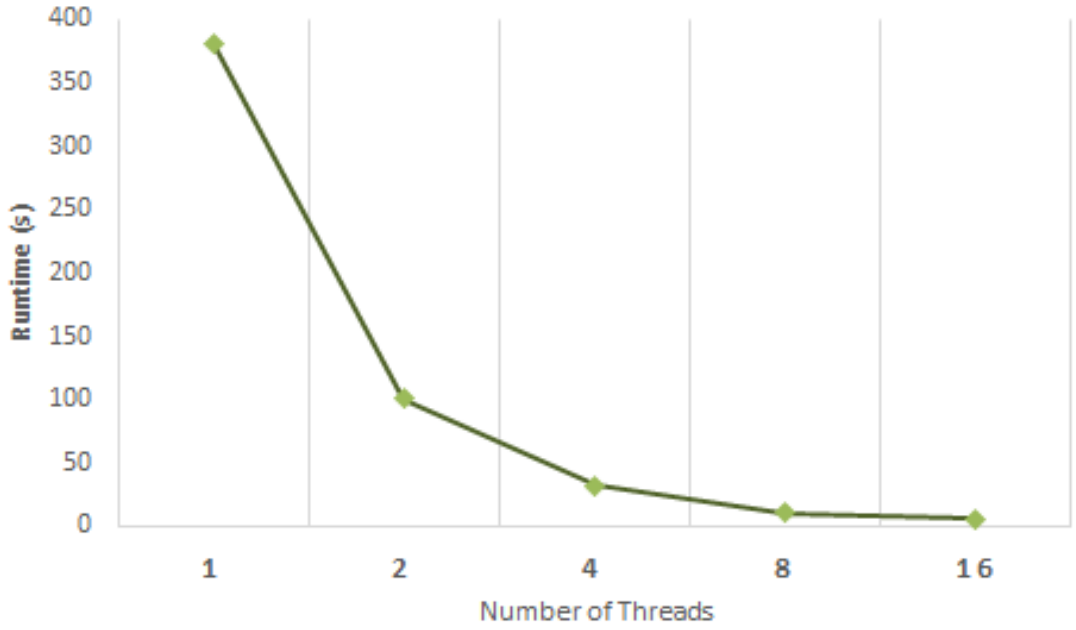


Figure 6.4: NA-STSJ-WS: vary number of threads

6.3.4 Vary Dynamic Ratio

We vary dynamic ratio for our NA-STSJ-WS algorithm. As mentioned in Section 5.5.1, the dynamic ratio is defined as the ratio of the number of data groups in the remote queue over that in the local queue. It can vary from 0 to 1, where 1 means that all the groups in the queue belong to remote queue, enabling every other node to access groups of every other node in the order of their topology, whereas 0 means all the groups belong in the local queue. In Figure 6.5, we observe that as the dynamic ratio is increased, the runtime decreases first and then increases slightly.

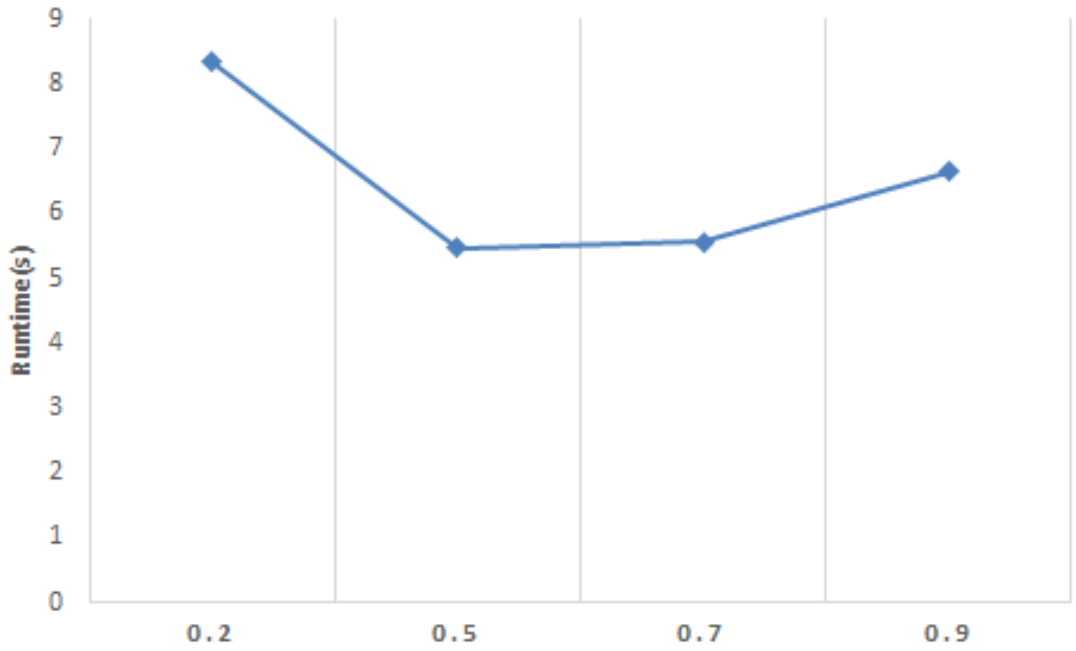


Figure 6.5: Vary Dynamic Ratio.

At around 0.5, the latency is minimum, which indicates that the optimum value of dynamic ratio. This is due to the fact that some nodes are able to finish tasks on their local queue first and when a significant number of threads become free, they can steal work from the still busy threads on other nodes.

6.3.5 Vary Thresholds

Next, we vary the spatial and textual thresholds for each of our algorithms. Figure 6.6 depicts the runtimes of NU-STSJ-B, NA-STSJ-BP and NA-STSJ-WS, as we vary spatial/textual thresholds. We use 0.5/0.8 as our default threshold values, which indicates that the should be within 0.5 km distance and must have a Jaccard's similarity of 0.8 or lower. The values are rough estimates for document closeness in a real world scenario, however, data can be highly skewed for dense regions, such as major cities. We observe that NA-STSJ-WS performs better in all cases but overall latency increases for all algorithms. This is due to the increase in the number of

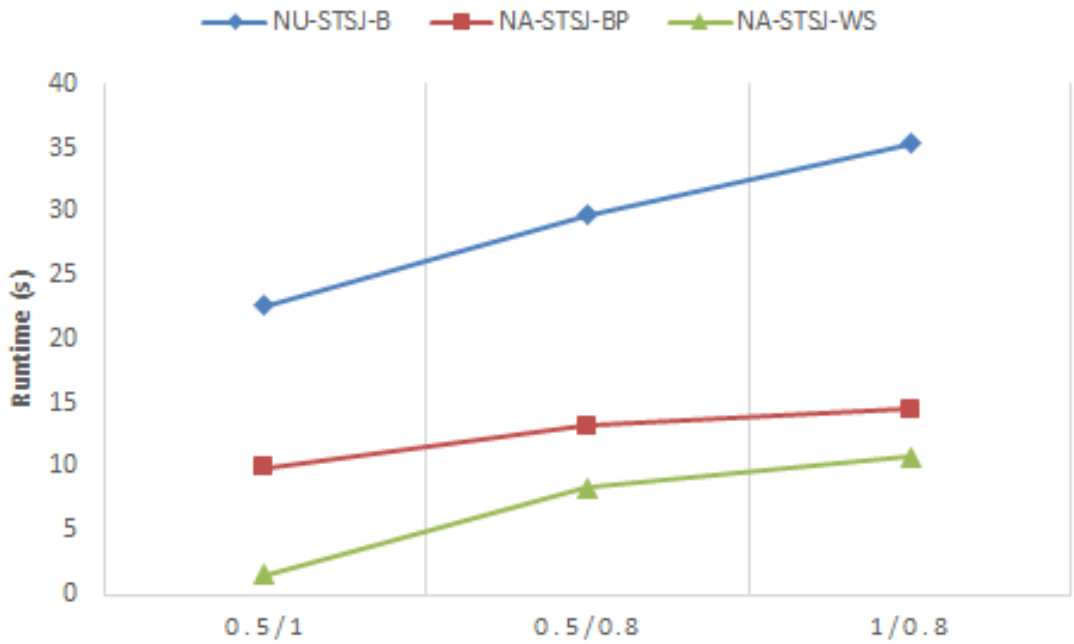


Figure 6.6: Vary spatial/textual thresholds.

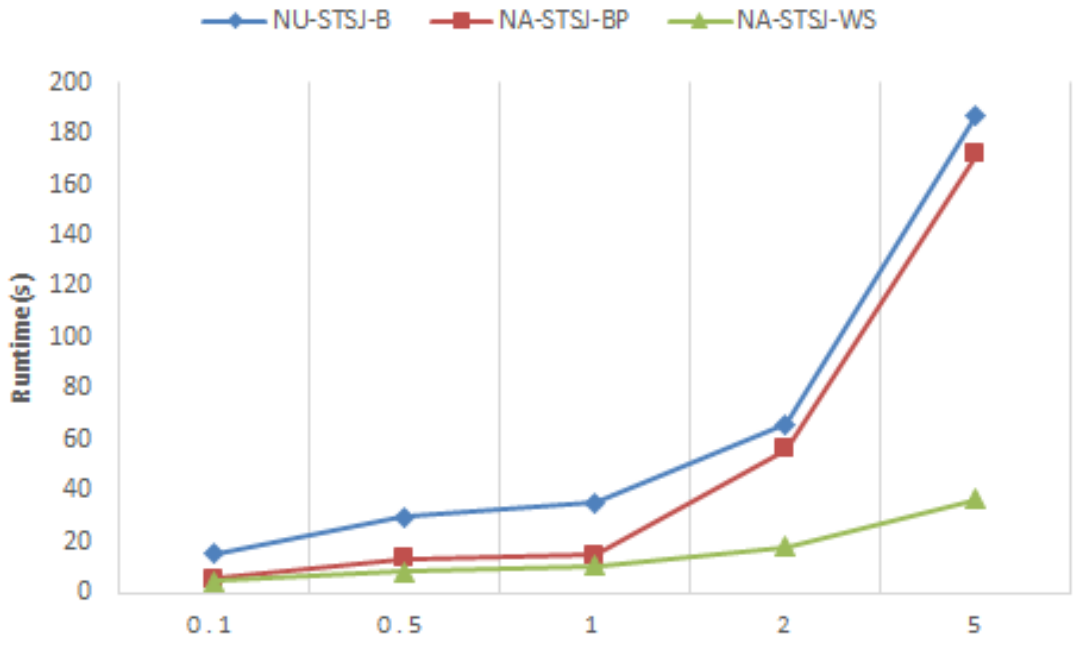


Figure 6.7: Vary spatial thresholds.

documents in each group. In the case of NU-STSJ-B, there is a sharp rise in runtime at 0.5/0.8 from that at 0.5/1. This is due to the lower textual threshold, during

runtime, a documents take more time in the PPJ join process, as prefix generation in case of lower textual threshold can be higher. Also, the poorer locality of groups in NU-STSJ-B results in some groups taking more time. In case of NA-STSJ-BP, the groups are distributed evenly, especially larger groups, which may result in less processing time overall and a more balanced performance. We also vary spatial threshold for large values. Figure 6.7 Shows the performance of algorithms by keep textual threshold as fixed to 0.8 and varying spatial threshold.

6.3.6 Self Join

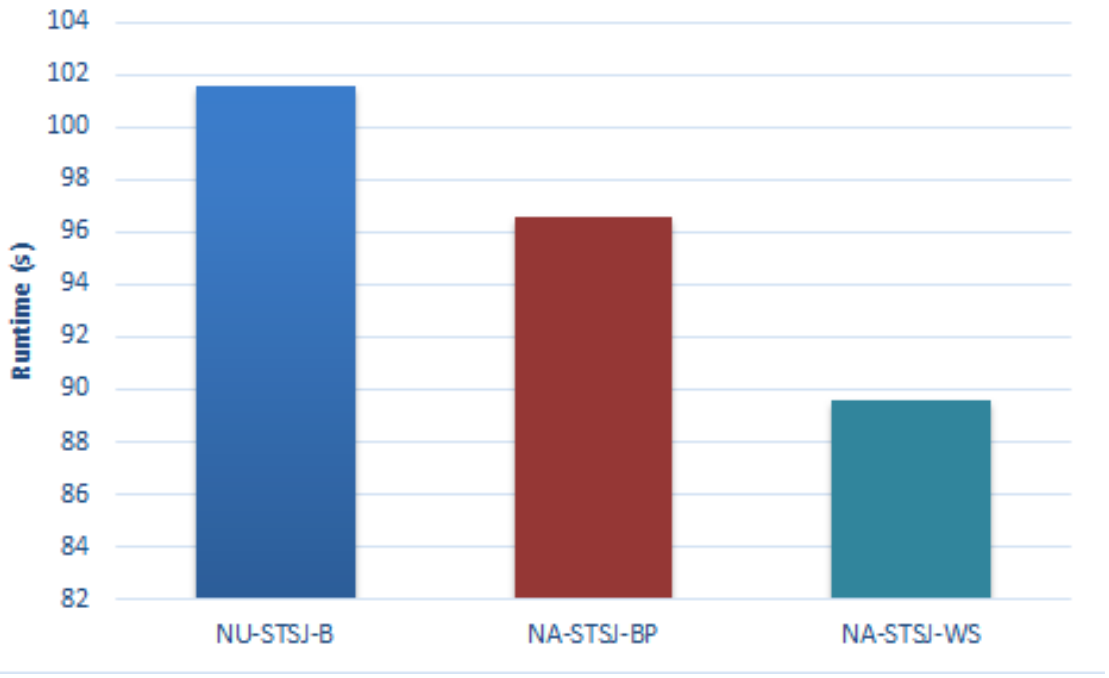


Figure 6.8: Self Join - vary algorithms.

In the case of self join, we join our default build dataset Tweets2mi with itself. The self join case is important, since all the documents in such a case have at least a corresponding document for any given spatial and textual threshold. In Figure 6.8 it can be seen that NA-STSJ-WS performs the best in the case of self join as well. Although, in this case, larger groups are formed, but still our algorithms performed

well. This is mainly due to the better locality of documents. In this case, the skewed documents are distributed evenly among all NUMA nodes, thus reducing load on any one particular node. The speedup is relatively low compared to join between the Spaten and Tweets2mi datasets, since the number of groups increases significantly for self join. NA-STSJ-WS performed about 1.1× better than NU-STSJ-B and about 1.05× times better than NA-STSJ-BP.

6.3.7 Build Time

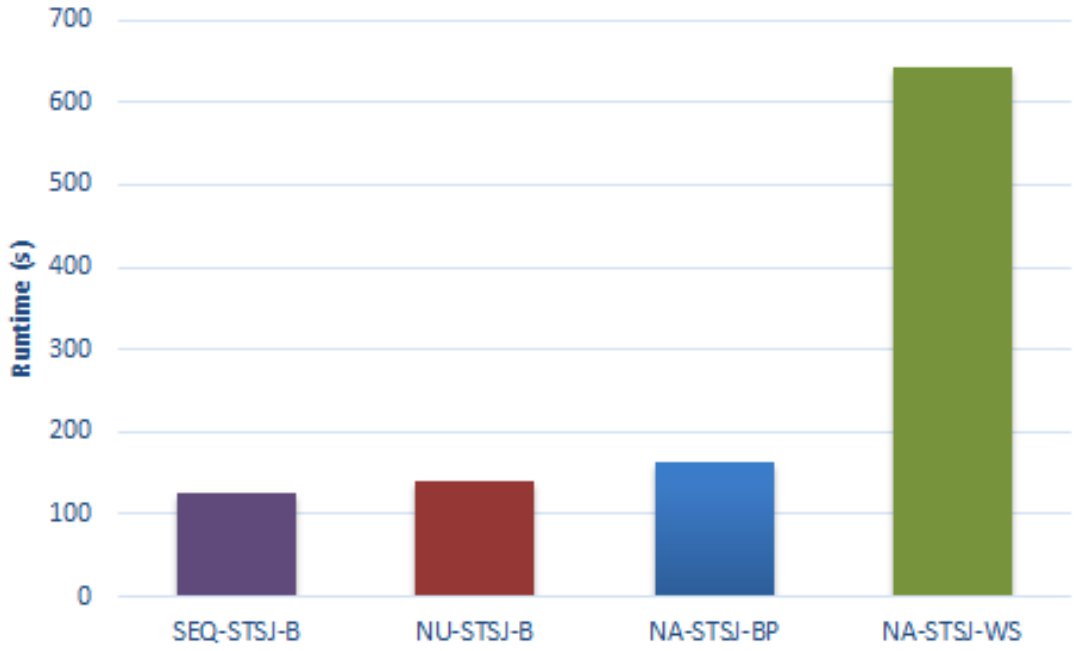


Figure 6.9: Build times with the algorithms.

Figure 6.9 represents the build times for different algorithms. By build time, we mean the total time taken by the algorithm from partitioning data to forming groups, and then allocating group data to a node. In the case of NA-STSJ-Q Algorithm 3 used in NA-STSJ-WS, procedures *partition_data*, *form_groups*, *create_group_on_numa* are part of the build process. Indexing time and join time are excluded from the build time. The baseline version of the algorithm builds up quickly as compared to the

others. This is expected, as the sequential version does not partition the spatial domain to categorize the documents into groups, but only creates inverted index once for whole dataset and follows the PPJ join as in Algorithm 1. NU-STSJ-B and NA-STSJ-BP first create groups based on cell and neighbouring cell based on spatial partitioning. While, NA-STSJ-WS takes the maximum time, which is due to the quadtree spatial partitioning and forming groups by range query.

6.3.8 Index Time

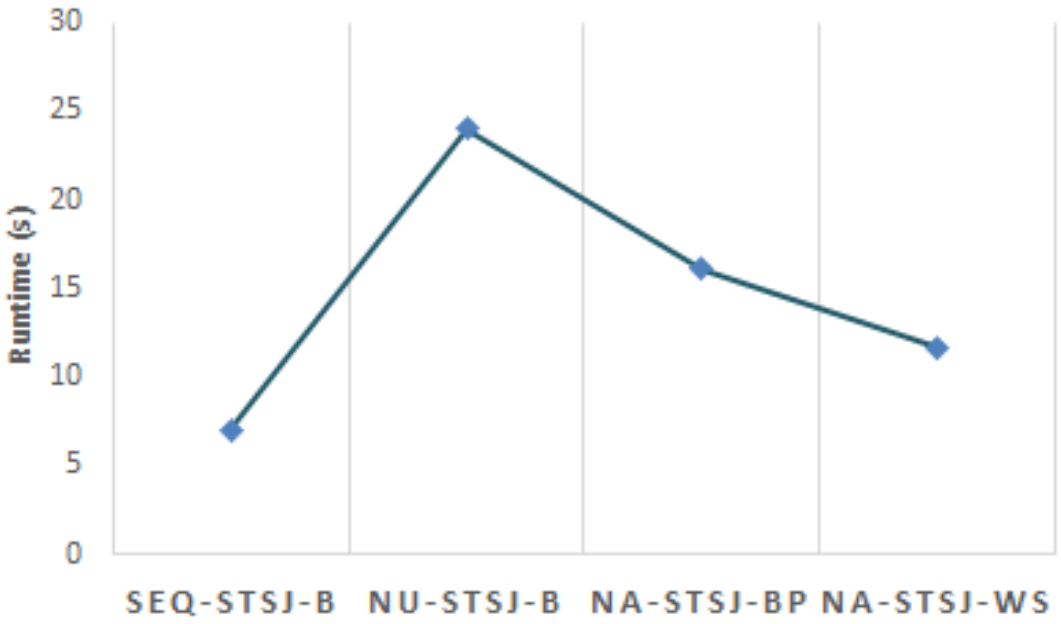


Figure 6.10: Algorithm index construction times.

Figure 6.10 shows the index times of various algorithms. Index time includes the time it takes for the algorithms to build the inverted index. SEQ-STSJ-B builds a single index, whereas for NA-STSJ-BP, it is built on a per group basis. In the case of NA-STSJ-WS the index is built on a per node basis. Figure 6.10 depicts that the index times slightly increase in the case of NU-STSJ-B as compared to SEQ-STSJ-B. This is due to the fact that each data group in the parallel baseline (non-NUMA) version is formed by combining their neighbouring cells, and the index is constructed for

each group. Thus, the index construction cost depends upon the number of groups formed. In the case of NA-STSJ-WS, a more global and NUMA localized index is created, thus saving largely on index construction. Also, it is interesting to note the runtime difference between NU-STSJ-B and NA-STSJ-BP, since the composition of the groups remains the same in both cases. The faster index time of NA-STSJ-BP indicates the NUMA locality of groups, while NU-STSJ-B have threads assigned based on Linux First touch policy. NA-STSJ-BP binds each thread to particular groups in the NUMA nodes they are located. Thus, NUMA locality greatly benefits the index construction cost in NA-STSJ-BP and NA-STSJ-WS.

6.3.9 Vary NUMA Policy

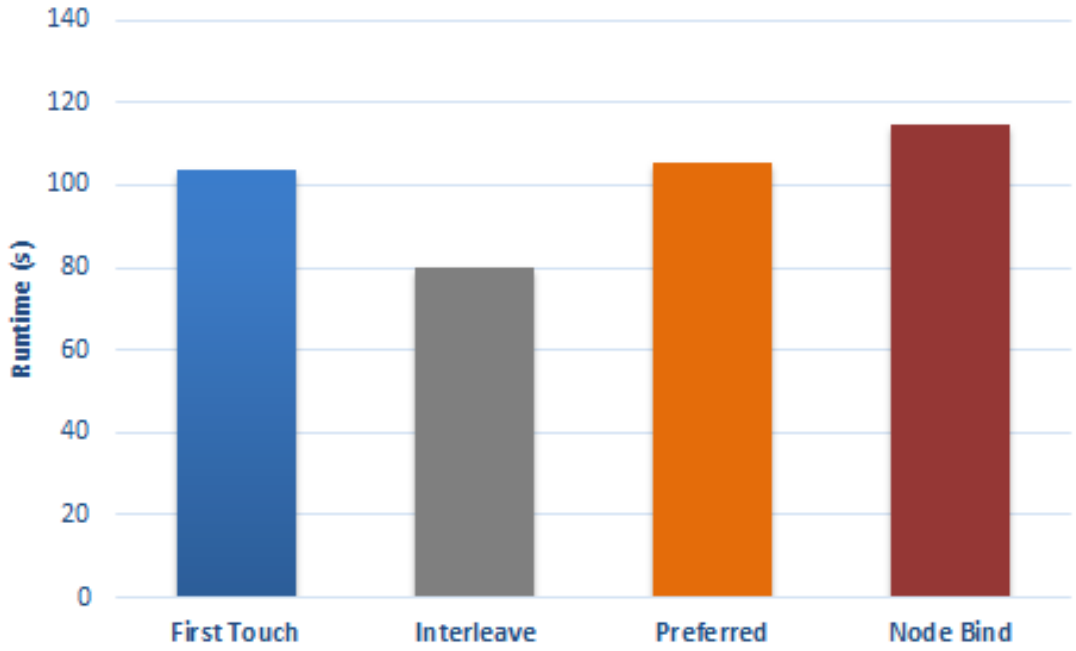


Figure 6.11: Vary NUMA Policies

We vary NUMA policies for NU-STSJ-B in the case of self join with the Tweets2mi dataset. Besides the Linux default First touch policy, we ran our programs with other policies like *numactl* Interleave policy, which spreads data of a program across all

possible NUMA nodes. Also, we bind our application to a single node in Bind policy, by binding CPU and memory to a single node and preferred policy as discussed in 2.7. As Figure 6.11 shows, the Interleave policy is the best case for this particular scenario. As data is spread across nodes evenly, Interleave performs better than others. In the Node Bind case, however, since data is placed on one node, every thread accesses data from the same node's memory, resulting in congestion during runtime. Therefore, Node Bind does not perform better than First touch. Preferred policy performs better than Node Bind but is slower than default first touch and interleave.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

With the rising volume of data that contain both spatial and textual components, novel approaches to spatio-textual query processing is becoming ever more important. Spatio-textual similarity join (STSJ) has been gaining in prominence in recent years. Due to its compute and data intensive nature, efficient parallelization of STSJ is necessary for its wide adoption. Modern machines adopt a NUMA architecture that entail varying memory access latencies depending on the distance between any two NUMA nodes. However, researchers have not yet considered how to parallelize STSJ while taking NUMA effects into account.

In this work, we have proposed three NUMA-aware parallel approaches to spatio-textual similarity join problem. Our first algorithm, NA-STSJ-BP considers NUMA-aware load-balancing by utilizing a bin packing of the spatial partitions. Our second algorithm, NA-STSJ-Q additionally considers skew-resilience. These two algorithms performs static assignment of tasks to the NUMA nodes. Our third algorithm, NA-STSJ-WS performs an adaptive placement of data and tasks using a topology-aware work stealing. This algorithm is able to account for the dynamic load conditions

in a system at runtime while executing STSJ. We have built a complete system in which we implemented our NUMA-aware algorithms. Furthermore, we implemented a sequential and a parallel baseline algorithm that are not NUMA-aware.

We evaluated our approaches with real-world datasets. Our evaluation suggests that our proposed NUMA-aware algorithm perform significantly better than both the sequential and parallel baseline approaches. In the best case, our third NUMA-aware algorithm, NA-STSJ-WS performs about **83**× better than the sequential baseline STSJ algorithm.

7.2 Future Work

Our current implementation uses Jaccard's similarity for text similarity. As a future work, it can be extended to support other similarity measures. We would also like to develop a more robust NUMA-aware version of STSJ. Since, existing STSJ techniques do not handle updates of spatio-textual objects, they require building all phases of the join algorithms to incorporate updated data. In future work, we would like to improve our system so that it can deal with updates efficiently.

References

[1] *Hilbert's space-filling curve*, pp. 9–30, Springer New York, New York, NY, 1994.

[2] *Numa policy library*, 2020, [Online; accessed 2020].

[3] *Quadtree, Wikipedia, the free encyclopedia*, 2020, [Online; accessed 2020].

[4] *Sørensen–dice coefficient Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=S2020, [Online; accessed 18-April-2020].

[5] *Z-order curve Wikipedia, the free encyclopedia*, 2020, [Online; accessed 2020].

[6] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik, *Efficient exact set-similarity joins*, Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06, VLDB Endowment, 2006, p. 918–929.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, *The r*-tree: an efficient and robust access method for points and rectangles*, Proceedings of the 1990 ACM SIGMOD international conference on Management of data, 1990, pp. 322–331.

[8] Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Christophides Vassilis, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, *Advances in database technology-edbt 2004: 9th international conference on extending*

database technology, heraklion, crete, greece, march 14-18, 2004, proceedings, vol. 2992, Springer, 2004.

[9] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis, *Spatio-textual similarity joins*, Proc. VLDB Endow. **6** (2012), no. 1, 1–12.

[10] Ruthie Bowles, *Packing algorithms: How they help warehouse operations*, 2020, [Online; accessed 2020].

[11] S. Chaudhuri, V. Ganti, and R. Kaushik, *A primitive operator for similarity joins in data cleaning*, vol. 2006, 05 2006, pp. 5 – 5.

[12] Quan Chen, Minyi Guo, and Haibing Guan, *Laws: Locality-aware work-stealing for multi-socket multi-core architectures*, Proceedings of the 28th ACM International Conference on Supercomputing (New York, NY, USA), ICS '14, Association for Computing Machinery, 2014, p. 3–12.

[13] Wikipedia contributors, *Space filling curve*, 2020, [Online; accessed 2020].

[14] M Corrêa, R Chanin, A Sales, R Scheer, and AF Zorzo, *Multilevel load balancing in numa computers*, 20th ACM Symposium on Operating Systems Principles (SOSP 2005), 2005, pp. 1–9.

[15] Anusuya Datta, *How are digital technologies influencing geospatial technology trends?*, 2019.

[16] M. Diener, E. H. M. Cruz, and P. O. A. Navaux, *Locality vs. balance: Exploring data mapping policies on numa systems*, 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, March 2015, pp. 9–16.

[17] Thaleia Dimitra Doudali, Ioannis Konstantinou, and Nectarios Koziris, *Spaten: A spatio-temporal and textual big data generator*, IEEE BigData, 2017.

[18] Ulrich Drepper, *What every programmer should know about memory*, 2007.

[19] Raphael Finkel, *Quad trees: A data structure for retrieval on composite keys.*, Acta Inf. **4** (1974), 1–9.

[20] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth, *Challenges of memory management on modern numa systems*, Commun. ACM **58** (2015), no. 12, 59–66.

[21] Antonin Guttman, *R-trees: A dynamic index structure for spatial searching*, SIGMOD Rec. **14** (1984), no. 2, 47–57.

[22] Jiawei Han, Micheline Kamber, and Jian Pei, *2 - getting to know your data*, Data Mining (Third Edition) (Jiawei Han, Micheline Kamber, and Jian Pei, eds.), The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, Boston, third edition ed., 2012, pp. 39 – 82.

[23] Piotr Indyk and Rajeev Motwani, *Approximate nearest neighbors: towards removing the curse of dimensionality*, Proceedings of the thirtieth annual ACM symposium on Theory of computing, 1998, pp. 604–613.

[24] Sitong Liu, Guoliang Li, and Jianhua Feng, *Star-join: Spatio-textual similarity join*, Proceedings of the 21st ACM International Conference on Information and Knowledge Management (New York, NY, USA), CIKM '12, Association for Computing Machinery, 2012, p. 2194–2198.

[25] Daniel Müller, *Memory and thread management on numa systems*, 2013.

[26] Jinfeng Rao, Jimmy Lin, and Hanan Samet, *Partitioning strategies for spatio-textual similarity join*, Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial '14, 11 2014, pp. 40–49.

[27] Leonardo Andrade Ribeiro and Theo Härder, *Generalizing prefix filtering to improve set similarity joins*, Inf. Syst. **36** (2011), no. 1, 62–78.

[28] Hanan Samet, *The quadtree and related hierarchical data structures*, ACM Comput. Surv. **16** (1984), no. 2, 187–260.

[29] Venu Satuluri and Srinivasan Parthasarathy, *Bayesian locality sensitive hashing for fast similarity search*, Proc. VLDB Endow. **5** (2012), no. 5, 430–441.

[30] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos, *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*, Proceedings of the 13th International Conference on Very Large Data Bases (San Francisco, CA, USA), VLDB '87, Morgan Kaufmann Publishers Inc., 1987, p. 507–518.

[31] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey, *Streaming similarity search over one billion tweets using parallel locality-sensitive hashing*, Proc. VLDB Endow. **6** (2013), no. 14, 1930–1941.

[32] https://developer.twitter.com/en, 2018.

[33] Jiannan Wang, Guoliang Li, and Jianhua Feng, *Can we beat the prefix filtering? an adaptive framework for similarity join and search*, Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '12, Association for Computing Machinery, 2012, p. 85–96.

[34] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang, *Leveraging set relations in exact set similarity join*, Proceedings of the VLDB Endowment (2017).

[35] Wikipedia contributors, *Jaccard index — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Jaccard$_i ndexoldid = 945587689, 2020, [Online; accessed 18 - April - 2020]$.

[36] Yiming Xiang, Yi Zhuang, and Nan Jiang, *An efficient block index scheme with segmentation for spatio-textual similarity join.*, TIIS **11** (2017), no. 7, 3578–3593.

[37] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang, *Efficient similarity joins for near-duplicate detection*, ACM Trans. Database Syst. **36** (2011), no. 3.

[38] Yu Zhang, Youzhong Ma, and Xiaofeng Meng, *Efficient spatio-textual similarity join using mapreduce*, Proceedings of the 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 01 (USA), WI-IAT '14, IEEE Computer Society, 2014, p. 52–59.

[39] Mingdong Zhu, Derong Shen, Ling Liu, and Ge Yu, *Hybrid-lsh for spatio-textual similarity queries*, APWeb, 2015.

Vita

Candidate's full name: Saransh Gautam
University attended: B.Tech, 2014, Harcourt Butler Technological Institute affiliated to Uttar Pradesh Technical University, Kanpur, UP, India
Publications: NA
Conference Presentations: NA