Vebjørn Ohr

NASH

Range Search over Temporal, Numerical, and Geographical Annotated Documents

Master's thesis in Computer Science
Supervisor: Dhruv Gupta
June 2022

Master's thesis

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

NTNU
Norwegian University of
Science and Technology

Vebjørn Ohr

NASH

Range Search over Temporal, Numerical, and Geographical Annotated Documents

Master’s thesis in Computer Science
Supervisor: Dhruv Gupta
June 2022

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

Abstract

Text documents such as news and encyclopedia articles often contain tremendous amounts of information about real-world events. However, there is a challenge to extract the most relevant snippets of information from millions of lengthy documents. Semantics can be extracted as annotations from the text using Natural Language Processing tools. These annotations can contain entities such as persons, organizations, and locations, as well as temporal and numerical expressions. Moreover, some annotations describing numerical, temporal, and geographical expressions can represent uncertain information, e.g., the phrase "1990s" referring to a ten-year interval. This thesis presents an approach to support full-text semantic search over text documents and their extracted annotations, with capabilities of spatial restrictions on the uncertainty-aware interval annotations. This is presented as the indexing system NASH. The annotated documents are stored using inverted indexes in a layered data model. Each layer represents one annotation type, and all layers share the positional information of the tokens. To support efficient retrieval, the numerical, temporal, and geographical expressions are reduced to one dimension by using Z-order curves, which translate the two-dimensional values into one-dimensional hashes. These hashes are stored as part of the layered index to allow for full-text semantic search with spatial query capabilities. Different optimizations are implemented to make the range-based queries over the Z-order curves more efficient. The system is evaluated using a set of semantic range queries, measuring the time used from query creation to the posting list retrieval. The queries are executed using different configurations of range sizes and search precision over three annotated document collections of differing sizes. The results demonstrate a functioning system and indicate that the system scales well with increasing collection sizes. Slight differences in result-set sizes and execution times between levels of search precision indicate that the range search optimizations are working well, with a small loss of precision, while increasing recall of the search.

Sammendrag

Tekstdokumenter som nyhetsartikler og encyklopediartikler inneholder ofte store mengder informasjon om ekte hendelser. Det er likevel en utfordring å kunne hente ut de mest relevante delene av informasjon fra millioner av større dokumenter. Semantikk kan trekkes ut som annoteringer av teksten ved å bruke Natural Language Processing (NLP) verktøy. Disse annoteringene kan inneholde entiteter som personer, organisasjoner, og lokasjoner, så vel som tidsmessige og numeriske uttrykk. For øvrig kan noen annotasjoner som beskriver numeriske, tidsmessige, og geografiske uttrykk representere usikker informasjon, for eksempel frasen "1990-tallet" som referer til et ti-års intervall. Denne oppgaven presenterer en tilnærmingsmetode for å støtte fulltekst semantisk søk over tekstdokumenter og deres ekstraherte annotasjoner, med mulighet for romlige restriksjoner på usikre annotasjoner. Dette presenteres som indekseringssystemet NASH. De annoterte dokumentene er lagret ved bruk av inverterte indekser som en del av en lagvis datamodell. Hvert lag representerer en annotasjonstype, mens alle lagene deler posisjonsinformasjon for toknene. For å effektivisere søking etter numeriske, tidsmessige, og geografiske uttrykk, reduseres uttrykkene til én dimensjon ved bruk av Z-ordenskurver, som oversetter de to-dimensjonale verdiene til en-dimensjonale hasjer. Hasjene lagres som en del av den lagvise indeksen for å tillate fulltekst semantisk søk med område-baserte søkemuligheter. Forskjellige optimeringsmetoder brukes for å gjøre de område-baserte spørringene mer effektive over Z-ordenskurven. Systemet evalueres ved å bruke et sett av semantiske intervall-spørringer over numeriske og tidsmessige annotasjoner, og måler tiden det tar fra spørringskonstruksjon til listen av treff er mottatt. Spørringene kjøres med forskjellige konfigurasjoner av intervallstørrelse og søkepresisjon over tre annoterte dokumentsamlinger av forskjellige størrelser. Resultatene demonstrerer et funksjonelt system og indikerer at system skalerer godt med økende størrelse på samlingene. Små forskjeller i treffstørrelser og kjøretid mellom nivåer av søkepresisjon, indikerer at optimaliseringene for intervallsøket fungerer bra, med et lite tap av presisjon mens treffandelen til søket økes.

Contents

Chapter 1

Introduction

1.1 Motivation

The amount of text data available digitally has significantly grown over time. Document collections such as news archives and encyclopedias contain detailed information on real-world events from both the recent and historical past. Many text analysis tasks require searching over such documents to extract relevant event-oriented information. This includes searching for relevant text regions which contain temporal, numerical, and geographical expressions. The text may additionally mention named entities such as persons, and organizations. Advancements in Natural Language Processing (NLP) have made it possible for computers to interpret deeper meanings in text than what is represented by the words themselves. By extracting the entity mentions and combining them with traditional full-text search, greater capabilities can be achieved when searching for event-oriented queries.

Important events contained in news archives and encyclopedic resources form an important aspect of our collective historical past. Being able to query such large collections of factual documents efficiently can be of great importance for journalists, historians, and lawyers, that require the capability to extract relevant sections from the text. Existing full-text search engines enable searching for combinations of keywords and text phrases to retrieve matching documents. If a user searches for the phrase "European wars during the 18th century", the search engine will retrieve documents matching these words in some manner. However, a document mentioning "The Seven Years' War (1756-1763) was a global conflict between Great Britain and France for global pre-eminence." is less likely to be a good match for the query as very few words match between the document and the query. Even though the user wants results mentioning wars during the 100-year span from 1700 to 1799, the search engine only matches the search phrase "18th century" with no understanding of an actual temporal range. The words *Britain* and *France* can be stored as word tokens in the search engine index, but there is no knowledge present that indicates that these tokens represent locations, or that they are part of the continent of Europe. It is clear that being able to extract such ranges for geographical, temporal, and numerical expressions will increase the capabilities of a text-oriented only search engine. Furthermore, by storing the positions of these annotations, a search engine can extract specific text regions which match the query.

Executing spatial queries over geographical data is a well-researched area. Customised database systems, called Geographic Information Systems (GIS) [1], are developed and optimized to search for such data. These systems often utilise special indexing techniques to support very specific operations and are thus not compatible with other types of data, nor applicable in a full-text search context. However, a spatial indexing technique called Geohashes [2] represents points on Earth at varying accuracy, as a single string of text using space-filling curves [3]. This means that the two-dimensional data is transformed into a one-dimensional representation while still supporting spatial queries. Because the data is one-dimensional, simple inverted index structures can be utilised to search the geographical data related to documents. This motivated the research into how a similar approach could be used to index numerical and temporal interval expressions, and thus allow for spatial queries over time, numbers, and locations, as part of a semantic full-text search system.

1.2 Contributions

The work of the thesis explores how documents and accompanying annotations can be indexed to support semantic full-text search. Semantic full-text search here refers to search allowing for positional matching over a combination of annotations and words. Multiple types of annotations can be indexed, but the main concern is the indexing of ambiguous, and multi-dimensional annotations such as geographical locations, temporal, and numerical expressions, that enable event extraction based on proximity in space and time. The focus of the work is on indexing and retrieval of temporal and numerical ranges, but the location annotations are included for completeness. The research question is formulated as follows:

> *How can full-text semantic search over large, annotated document collections be implemented, while supporting proximity-based range queries over numerical, temporal, and geographical annotations?*

This includes the design and implementation of a data model, query operations, and an index infrastructure. This will result in a functional search engine with the required range-based search capabilities over temporal and numerical annotations. The focus of the implementation is on the utilisation of the space-filling curve to achieve the desired range-based query operations.

The main contribution of this thesis is the indexing of multidimensional annotations, such as geographical, temporal and numerical expressions which in combination with words enables full-text semantic search. To do so, space-filling curves are used to transform such annotations to one-dimensional values, which can then be stored in an inverted index structure. The approach is implemented in the proposed index system called NASH. Specifically, the Z-order curve is used to support range queries over temporal and numerical intervals, by pre-computation a minimum set of covering Z-order values which cover the search range. To further speed up the query processing, a set of different optimizations for the Z-order range searches are implemented and evaluated.

Chapter 2

Related Work

This chapter starts by describing fundamental concepts for traditional full-text and semantic search. The remaining sections are devoted to describing underlying concepts and related work regarding indexing and querying of multi-dimensional data.

2.1 Information Retrieval

Information retrieval (IR) deals with the storage and access to informational items such as text and documents, with the intent to provide the user with easy access to the parts which are of interest [4]. This section briefly introduces some of the core concepts from the field of IR which are utilized in the research.

2.1.1 Document

A document in the information retrieval setting can be defined as a single unit of textual information [4]. The document is usually structured in some way following a syntax. This thesis uses the term document to describe the largest unit of data which is to be indexed and queried, such as news articles or entries in an encyclopedia.

2.1.2 Indexing

One of the most basic ways of searching documents is through sequential search, reading each document one after, word after word, looking for conditions matching the query. This works fast when all the documents fit in main memory, however, when document collections outgrow the available memory in the system, sequential scans over the data are limited by the speed of the secondary storage. To make the search more efficient, indexes are needed. The index speeds up the search by storing the most relevant information of each document in optimized data structures and uses compression techniques to decrease the storage space needed. Most of the classical IR models index documents based on their words or groups of consecutive words, called index terms. This can be all individual words present in the document, or the words that best describe the document based on some measure such as frequency distributions.

The Inverted Index

The inverted index is the most used indexing technique to speed up query processing [4]. Figure 2.1 illustrates an inverted index. It consists of two elements: vocabulary and occurrences. For basic text indexing, the vocabulary is the set of all different words appearing in the text, and the occurrences, or posting lists, are lists of documents (represented by document IDs) in which the term occurs. The terms of the vocabulary can be determined by some tokenization process which breaks the text into separate regions of text. For example, words can be defined as a sequence of letters followed by a separator (e.g., a whitespace). The vocabulary terms can however be any type of information related to the documents, such as extracted annotations. In addition to document IDs, the posting list can contain positional information of where in the document the term occurs. For example, Figure 2.1 shows that the term “lorem” appears in document four at position one, and in document 12 at positions 42 and 120.

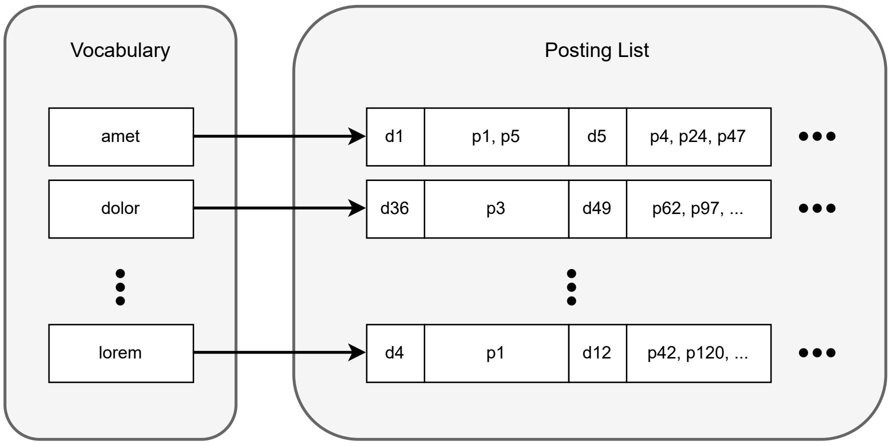


Figure 2.1: The vocabulary with terms on the left, points to document entries as posting lists to the right. The document entries have token positions stored to support positional queries.

2.1.3 Searching the Inverted Index

The inverted indexes can be used to support full-text search. The user formulates a query of index terms, and the search engine looks up each query term in the index to retrieve posting lists of relevant documents.

Word Queries

Single terms can be directly looked up in the index and the posting list of occurrences is retrieved. If the query consists of multiple terms, the posting lists of each term can be joined to find the relevant documents for conjunctive queries or merged for disjunctive queries. The cost of looking for a word in the index is dependent on which data structure is used to implement the index. Data structures such as the B-trees, hashing, and tries give $O(m)$ search cost, where m is the number of terms in the search query [4]. There is also a cost of joining the posting lists, especially if the search terms are present in many documents resulting in large posting lists.

Context Queries

To search for multiple words in a specific order, or that are close to each other. Phrase queries can find matches based on a sequence of words. This can be implemented by storing positional data for each word in the inverted index, and verifying that the positions are in the correct order. The phrase query can also be relaxed to a proximity query, where there can be a set number of words or characters between the search terms which do not match.

Range and Prefix Queries

More advanced queries such as prefix and range search can also be supported by an inverted index. Prefix queries match documents based on if words in the document start with the search term. This type of search can be made more efficient by indexing prefixes of each word (e.g., *dog* is indexed with tokens *d do dog*). Range search retrieves all documents with words which lie between two search terms in lexicographical order. The efficiency of this can be dependent on the index data structure, and how the terms are ordered when stored.

2.1.4 The Boolean Model

IR systems utilise different models to determine which documents should be retrieved. The three classic retrieval models are the Boolean, the vector, and the probabilistic model [4]. The implementation of this research is only concerned with if a document matches or if it does not, i.e., there is no ranking of the documents or partial matching. The Boolean model then fits this requirement well, as it only retrieves documents fully matching the query.

The Boolean model is a simple model based on set theory and Boolean algebra. The model considers each index term as either present or absent in a document, i.e., a binary representation. The query language consists of three connective operators: *not, and, or*. A query can be built up of a combination of these Boolean operators between terms (t_n) or other sub-queries. The AND ($t_1 \wedge t_2$) operator is used for conjunctive queries. Each term is looked up in the index, and the posting list for each matching term is intersected, resulting in documents containing all these terms being returned. The OR ($t_1 \lor t_2$) operator is used in disjunctive queries, returning documents with at least one of the terms present in the inverted index. The posting list for each matching term is merged. The NOT ($\neg t_1$) operator returns documents which do not contain the query term. All these three operators can be combined to allow for more advanced queries.

For example, to search for a query such as $David \land Hilbert \land paper \land \neg 1891$, the query processor looks up posting lists for each search term in the inverted index and returns those documents mentioning *David Hilbert paper*, and not the word *1891*. If positional data is stored in the index, the order of the terms can be restricted as well.

2.1.5 Retrieval Evaluation

There are different ways of evaluating an IR system based on the produced results. Two of the most widely used metrics to evaluate retrieval quality are recall and precision. These metrics are defined as follows: where, A is the set of retrieved documents based on the query, and R is the total set of all relevant documents in the database:

$$\text{Precision} = p = \frac{|R \cap A|}{|A|} \tag{2.1}$$

$$\text{Recall} = r = \frac{|R \cap A|}{|R|} \tag{2.2}$$

Precision is the fraction of the retrieved documents which are of relevance, and recall is the fraction of all relevant documents in the database which were retrieved. E.g., receiving 50 documents from the IR system, of which 20 are relevant equals a precision of $p = \frac{20}{50} = 0.4$. If five other relevant documents exist but were not retrieved, the recall is given by $r = \frac{20}{25} = 0.8$. The retrieved documents can also be ranked by some metric before being presented to the end-user, but for the Boolean model, a ranking between the results is meaningless as all documents must match the query.[1]

2.2 Semantic Search

The term semantic search has different definitions depending on the context but can be defined as "search with meaning" [5]. In this thesis, it is used to define full-text semantic search. This implies an inverted index model to search for text phrases in documents, but with the additional capability of interpreting semantic

[1]For disjunctive queries (OR), you could increase the score based on how many of the disjunctive terms matched, but this is not utilised in this thesis.

information of the text. The semantic information can be represented by annotations extracted from document contents.

2.2.1 Natural Language Processing

Documents contain natural language text that can easily be understood by humans, but not by machines. Nowadays, Natural Language Processing (NLP) tools can be used to interpret the semantics of words, allowing machines to obtain a deeper understanding of the document contents. With the ability to annotate documents using NLP tools, one can automate rule-based information extraction.

NLP tools can do basic transformations of the text data, such as tokenization, splitting the tokens into sentences, lowercasing the tokens, removing stop words, lemmatization, etc. Additionally, more advanced tasks such as Part-of-Speech (PoS) tagging, sentence parsing, Named Entity Recognition (NER), and constructing word vectors can be done [5]. Many of these transformations can be used in combination with the indexing of words to increase the number of relevant documents retrieved.

One of the NLP tasks that is important for the work in this thesis, is NER. NER recognizes word sequences in the text which constitute an entity (usually a proper noun). An entity can be a person, organization, location, etc. The NER process can also be linked to a knowledge base with synonyms and variations of the entity names, as well as the entity's relations to other entities. More advanced NER techniques can detect co-reference mentions, i.e., multiple word phrases in the same context can refer to the same entity. For example, in the sentence "Joe Biden is the president of the USA. He was elected in 2020", the term "he" refers to the previous mentioned named entity and can be determined automatically.

2.2.2 Implementing Semantic Search

Bast et al. [5], categorize research on semantic search into different categories based on the data type and search paradigm used. Data types *include text*, *knowledge bases*, and *combined data*. The search paradigms are *keyword*, *structured*, and *natural language search*. This thesis is concerned with keyword search over combined data, which in this case is a combination of text and entity annotations. The search also supports retrieving phrases as text regions. Knowledge bases [6] can add further information to the annotations, however, such implementations are out of scope for this thesis.

Existing Systems

Keyword search on combined data, or semantic full-text search, is often implemented using inverted indexes [5]. One way of doing this is by adding artificial words to the index which describes the annotation. For example, to mark the text "Alan Turing" as an entity, the phrase *entity:Alan_Turing* can be added to the index, which makes searching for entities as simple as looking up entries in the inverted index. Tablan et al. [7] add further artificial words such as *type:computer_-scientist* to be able to conduct more complex queries. If the extra tokens retain the positional information of the word token(s) they describe, annotations and text phrases can be combined in positional queries. For example, a query such as $type : computer\_scientist \wedge article \wedge 1930s$ can be formulated to match with all entities of the *computer_scientist* type. The Broccoli search engine [8] uses a more entity-focused implementation, receiving matching entities which occur in the document collection, without needing to retrieve the documents themselves. This is done by inserting entities in the inverted index for each *semantic context* it appears. A semantic context is a collection of words belonging together, such as a clause in a sentence. GYANI [9, 10] uses a layered data model of text and annotations, utilising inverted and direct indexes to store the positions of annotations, text tokens, and combinations of these, allowing queries with regular expressions over annotations and word sequences. They also implement direct indexes to store sentence boundaries such that queries can be restricted to text regions within sentences.

Existing approaches allow for entity matches which are mentioned close together in the text. However, none of the approaches allows for range-based queries over multi-dimensional geographical, temporal, and numerical expressions as part of the full-text search.

2.3 Multi-dimensional Data

Multidimensional data consist of observations having attribute values in more than one dimension. These values can be of the same or different types, both spatial and non-spatial. In this thesis, only spatial values of the same type are considered, which can be described in the two-dimensional (2-D) Euclidean space. To be able to retrieve documents based on the multi-dimensional points, the range of values in each dimension needs to have an ordering between them. For numerical attributes, they can be ordered by their value, and for text attributes a lexicographical ordering can be used.

2.3.1 Locational Data

For locational data, each attribute is of the same type and unit, the distance in space. Spatial values can be represented as part of a Cartesian coordinate system, where each attribute of the spatial value describes its position in the coordinate system of that dimension. For two dimensions, X and Y are often used as labels to describe the two axes. Figure 2.2 shows a visual representation of the two-dimensional points in a coordinate system. Point A represents the attributes $x = 1$ and $y = 2$. These spatial values allow for proximity-based queries [11] to find data which is similar to each other. To be able to query such objects efficiently, the data model used in indexing must be taken into consideration.

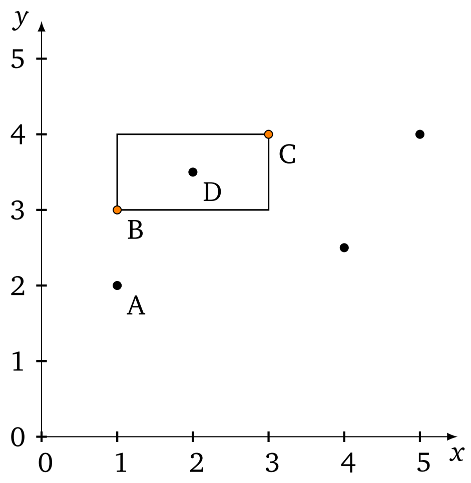


Figure 2.2: The figure illustrates a set of two-dimensional points and a geometry as part of a coordinate system.

The coordinate system can also be used to visualise geometrical shapes. A shape is in this case a two-dimensional area limited by a boundary of points. A closed polygon can be represented by an ordered list of these points, with the start point equalling the endpoint. To represent rectangles, only two points are needed: its minimum (l_1, l_2), and maximum (u_1, u_2) corners. Such a rectangle is called a Minimum Bounding Rectangle (MBR), as it describes the smallest rectangle which includes both points. In Figure 2.2, $B = (1, 3)$ is the lower bound of values and $C = (3, 4)$ is the upper bound of values of such a rectangle. The two other corners can be extracted from the minimum and maximum values.

2.3.2 Range and Interval

The term range can be used to describe the mathematical interval between two numbers which describes the upper (u) and the lower bound (l) of the interval.[2] The interval is the set of real numbers, where each number x satisfies the condition $l \leq x \leq u$. The inclusive interval is formulated as $[l, u]$, where the lower and upper bound are included in the interval. For example, a range of values from zero to ten is depicted by $[0, 10]$. A range can have multiple dimensions, and is depicted as two individual ranges representing each dimension, $[l_1, u_1], [l_2, u_2]$, or the ranges can be combined, $[l_1, u_1, l_2, u_2]$. Each range of the multi-dimensional range thus implies upper and lower bounds for a single dimension. There is a clear connection between two-dimensional intervals and MBRs. As the MBR defines upper and lower bounds, it directly translates to a two-dimensional interval.

The term "maximum range" is used to describe the largest possible range for a dimension, denoting the upper and lower bound of all sub-ranges within the

[2]Interval and range are used to describe the same concept; however, the term range is more common in IR and search.

dimension. That is, the maximum range represents the universe of possible values for a given dimension.

Range Queries

A range query searches over intervals, or ranges of values lying within an upper and a lower bound. This can be applied to a single, or multiple dimensions. For a two-dimensional search, any point which lies within the two ranges defined by $[l_1, u_1]$ $[l_2, u_2]$ is returned. This type of range search can be represented by the corresponding MBR. The MBR corners correspond to the lower (l_1, l_2), and upper (u_1, u_2) bounds of each dimension. In Figure 2.2, $B = (1, 3)$ is the lower bound of values and $C = (3, 4)$ is the upper bound of values of such a rectangle. This means any point where $1 \leq x \leq 3$ and $3 \leq Y \leq 4$ is contained in this rectangle and will be returned by the range search. That is, it is a 2-D search over intervals where $x \in [1, 3]$ and $x \in [3, 4]$. In this case, the point $D = (2, 3.5)$ is contained by the rectangle and is thus returned by the range search.

2.3.3 Spatial Queries

Chang [1] lists three basic spatial queries on geometries and points. A spatial query consists of a search geometry and a set of indexed geometries represented by one or more points. In this thesis, a search geometry is represented by an MBR with the points (x_l, y_l) and (x_u, y_u), denoting the lower and upper bounds of the X and Y in a two-dimensional range search. The indexed geometries can also be represented by an MBR, (x_1, y_1) and (x_2, y_2). Points can be indexed in the same manner with the restriction that $x_1 = x_2$ and $y_1 = y_2$.

Containment Queries Selects indexed geometries where the search geometry *contains* the object, or selects indexed geometries in which the search geometry is fully *contained by* . For a search MBR to contain a stored geometry, the smaller bounds of the indexed MBR must be larger than or equal to the search MBR's lower bounds, and the upper bounds of the indexed MBR must be smaller than or equal to the search MBR: $x_l \leq x_1$, $y_l \leq y_1$ for the lower values, and $y_2 \leq y_u$, $x_2 \leq x_u$ for the upper values. The contained by relation is the reverse, where it is the indexed MBR which must contain the search MBR.

Intersect Queries Selects geometries that intersect with the search geometry, meaning at least one of the indexed MBR corners are within the search MBR boundaries.

Proximity Queries Selects geometries that are close or adjacent to the search geometry. For MBR ranges, this can be achieved by increasing the size of the search MBR, i.e., subtracting from the lower bounds, and/or adding to the upper bounds, to cover a larger search area within a defined distance.

2.3.4 Indexing of Point Data

The inverted index can be applied to 2-D point data by constructing an inverted index for each of the attributes. Range queries over multiple attributes can then be done by doing a separate range query on the index of each attribute and intersecting the results. This is however wasteful, in the sense that two different indexes must be searched, which can result in an expensive intersection if the result set of each index is large. Other specialized data structures exist which are more efficient for searching multi-dimensional data points [11]. However, many of these approaches utilize indexing structures which are more complex in their implementations compared to the inverted index, and thus less compatible with full-text search. Such structures might also not be as readily available as the inverted indexing infrastructure in existing search engines and databases with other required capabilities in terms of scalability and performance.

Instead of introducing new data structures, another approach is to reduce the dimensionality of the document data. This way, existing data structures and algorithms optimized for one-dimensional data, such as the inverted index, can be utilised. Related to full-text semantic search, this means that the same indexing structure can be used for both one-dimensional and multidimensional annotations, as well as text.

2.4 Space-Filling Curves

One way of linearizing multidimensional values into a one-dimensional representation is through space-filling curves. G. Peano proved in 1890 [3] that there exists a curve passing through every point of a two-dimensional region (such as the unit square). These curves are called Peano curves, or space-filling curves. One year later, D. Hilbert recognized a general geometrical generating procedure that allowed for creating a wide variety of space-filling curves [3]. The space-filling curve also extends to n dimensions. Because the space-filling curve goes through every point of the n-dimensional region, it can be used to map multidimensional data to a location on the curve. Thus, any n-dimensional point in a region can be reduced to the position on the curve going through that region, resulting in a one-dimensional representation. This thesis is concerned with the applications in two dimensions, and further discussion will regard the space-filling curve in a two-dimensional space.

2.4.1 The Quadtree

The Quadtree [12] is a common technique used for spatial indexing. Even though the quadtree is not directly related to the space-filling curve, they share some characteristics which are helpful when explaining the space-filling curve in more detail. This includes how the space is divided into regions and how they both have hierarchical characteristics which can be visualised as a tree structure. Because of their similarities, space-filling curves can be used to construct quadtrees [13].

The quadtree is a generalization of the binary tree for treating data of two dimensions. A binary tree recursively splits a one-dimensional space in two, resulting in a maximum of 2^n nodes at each level, starting with $n = 0$ at the top level. In the quadtree each dimension is split in two, resulting in four total children, and thus 4^n nodes of each level. The region quadtree is a variant of the quadtree where the space is divided into equal size regions. Each region then represents a two-dimensional range, i.e., a minimum bounding rectangle. All points in the two-dimensional space will then belong to one of these regions, or buckets. Figure 2.3a shows how a two-dimensional space can be broken into smaller quadrants with the corresponding tree representation.

The tree representation also illustrates the hierarchical characteristics of the quadtree. The parent-child relationship of nodes means that all child nodes of the same parent are all contained by the parent MBR. In Figure 2.3, A is at level 0 (root) and represents the entire space. The next level breaks the space into four quadrants: B, C, D, and E. Each of these can be further divided into smaller quadrants, having the larger quadrant as its parent. Retrieving the root, A, would then return the entire space while retrieving K would return 1/16 of the entire space.

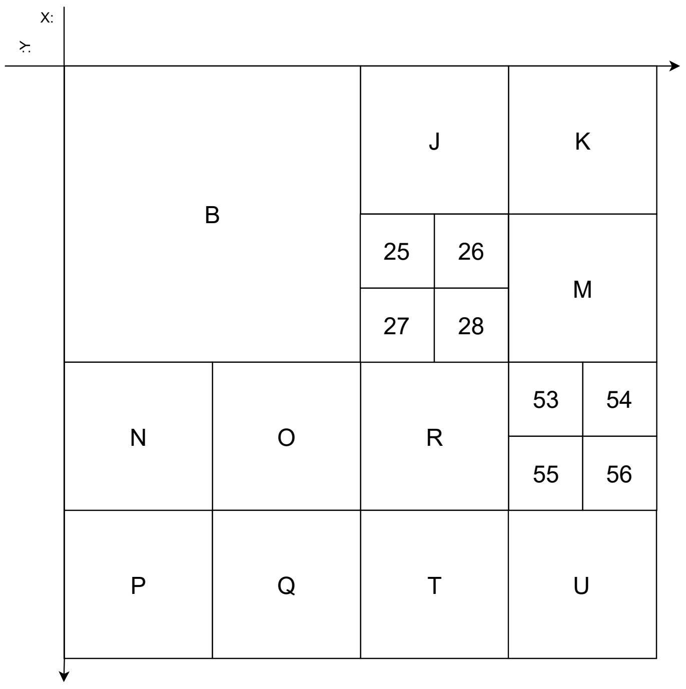
Each region can be described by their position in space. As each node can have four children, the position of the children within the parent region can be described by their intercardinal directions: northeast (NE), southeast (SE), southwest (SW), and northwest (NW). In Figure 2.3a, B is the northwest, C the northeast, D the southwest, and E the southeast quadrant of the common parent A.

2.4.2 Visualising Space-filling Curves

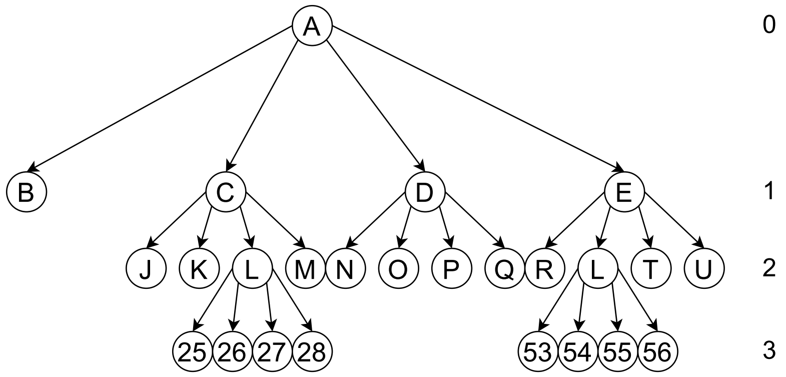
For space-filling curves, the two-dimensional space can be divided similarly to the region quadtree. The space is recursively divided into four equal size non-overlapping regions. The space-filling curve moves through these partitions in some manner, only moving in and out of a partition once. Each partition then has a unique position on the curve, and as the curve is linear the position can be represented by a one-dimensional value. Figure 2.4 illustrates an implementation of a space-filling curve where the space is divided once in each dimension, resulting in four regions. The curve goes through each region exactly once, giving a unique value for each region. Figure 2.5a partitions the dimensions four times each, resulting in 16 total partitions. The figure illustrates how each point in the coordinate system ($X, Y \in [0, 3]$) is mapped to a single region and a single point on the curve.

2.4.3 The Z-order Curve

One of the most used implementations of the space-filling curve is the Morton curve [14], also called the Z-order curve. The curve gets its name from the way the curve travels through the n-dimensional space in a ‘Z’ fashion, as shown in



(a) A two-dimensional space is broken down into recursively smaller quadrants.



(b) The tree representation of the same space.

Figure 2.3: The tree representation of the same search

Figure 2.4. This order can be achieved by bitwise interleaving the bits of X and Y which covers a region, starting with the most significant bit (MSB) of the Y-value. Each such unique bit-string represents a region in the 2-D space. The "curve" is thus a set of discrete values, akin to the quadtree regions or nodes. These values describing a partition are called Z-order values, or Z-order hashes.

Figure 2.4 illustrates how the Z-order hashes are calculated and how the order of the curve is determined. As the space is partitioned once in each dimension, two bits are needed to represent the position on the curve. The number of partitions is decided by the equation $p = 2^b$, where b is the bit length and p, the number of partitions. The northwest quadrant $(0, 0)$ has a 0 as the bit-value for both X and Y. Interleaving these two bits, starting with the Y bit, results in a hash of $00_2 = 0_{10}$ (the subscript show the number base) which indicates the first position. The next value on the curve is then (0, 1), as the X and Y bits of this point equals $01_2 = 1_{10}$.

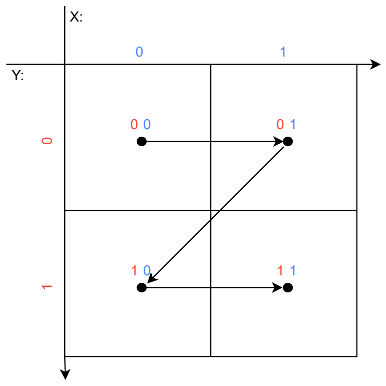
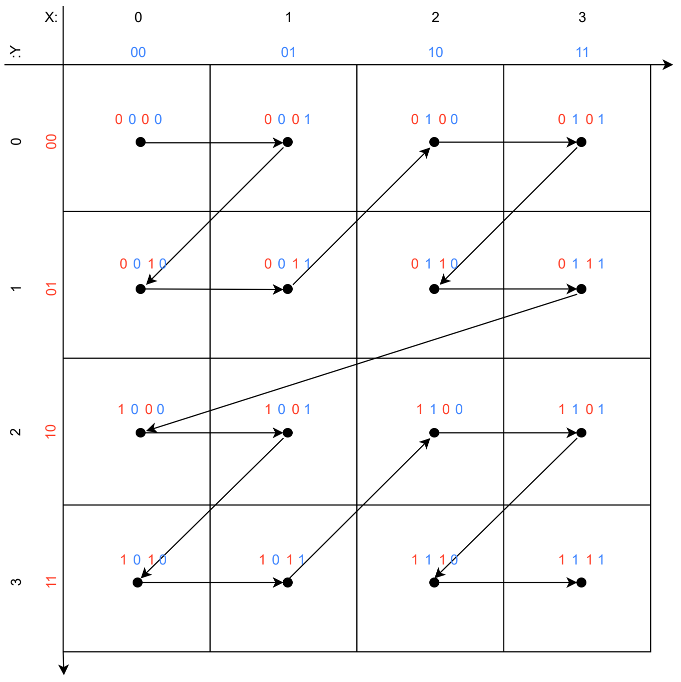


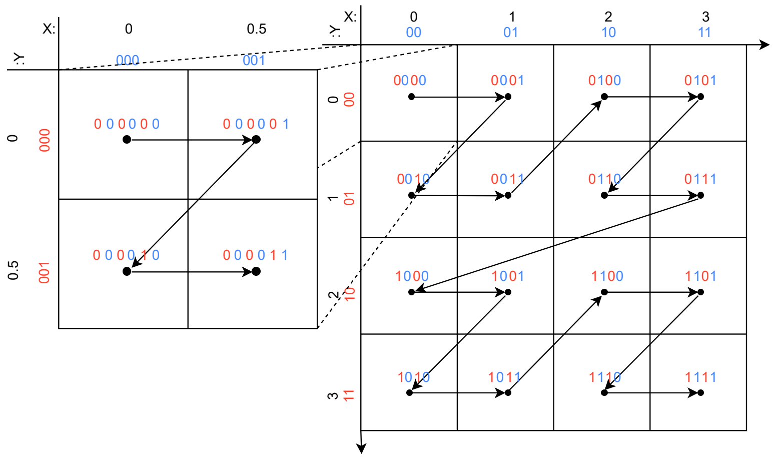
Figure 2.4: Z-order curve in two dimensions using 2 bits.

Figure 2.5a increases the number of bits by one for each dimension, to a total of 4 bits. The number of partitions is then increased to 16, hence the curve consists of 16 unique values. Comparing Figure 2.5a and Figure 2.4 makes the hierarchical properties of the Z-order curve apparent. Removing the last two bits of any 4-bit hash results in a position on the 2-bit curve. That is, the length of the hash determines which level of the tree representation the hash is a part of. Removing two of the rightmost bits equates to travelling up one level in the quadtree. Figure 2.5b illustrates travelling down the tree, by adding two bits to the hash in the rightmost position. Looking at the first partition in the northwest corner (0000_2), one can recursively partition it further into four new smaller quadrants by adding two bits to the Z-order representation.

There exist many other variants of space-filling curves which travel through space in different manners [15]. One such curve is the Hilbert curve [16], which like the Z-order curve can be seen in multiple applications. The Hilbert curve can give greater locality preservation but at the cost of more complex calculations [17].



(a) Implementation of a space-filling curve using Z-order in two dimensions with four bits.



(b) By increasing or decreasing the number of bits in the Z-order value, a smaller or larger area is covered.

Figure 2.5: Figures showing Z-order curves in two dimensions.

2.4.4 Hashing Points and Polygons to Z-order values

By utilising the Z-order curve on a coordinate system, each point in the coordinate system will belong to one of the Z-order partitions, or rather Z-order values. Figure 2.6a shows several points and a polygon in a coordinate system with $X, Y \in [0, 15]$. A four-bit z-order curve fills this space, resulting in 16 Z-order values. To calculate the Z-order value of a point, the value of each dimension (X and Y) is recursively compared against half the maximum range of its dimension. For example, the Z-order value of point C (11,6) can be calculated by first checking if the Y-value (6) is less than half of 15, if so the first bit is 0. Otherwise, if Y is equal to or larger than the mid-point, the first bit-value is set to 1. As 6 is less than 7.5, the first Y-bit is set to 0. Then the value is compared against half of 7.5, resulting in the second Y-bit being set to 1 ($6 \geq 3.75$). Similarly, the X-value is recursively compared against half the maximum X-range, resulting in X-bits of 1 and 0. The Y (01_2) and the X (10_2) values are then interleaved to give the Z-order value 0110_2. Polygons can be hashed by calculating the Z-order value of each point representing it. In Figure 2.6a the polygon F has four corners, and the area of the polygon span four different Z-order values. The polygon will therefore be broken down into a set of regions, or rather Z-order values, which cover the area. A greater bit-precision of the Z-order curve will result in a more precise representation by including more and smaller Z-order regions, but the region will always be built up of smaller rectangles, like pixels in a digital image.

2.4.5 Accuracy

As the Z-order curve naturally partitions the space into quadrants, a useful application is to utilise the Z-order curve as a bucketing tool similar to the quadtree, where each non-overlapping partition functions as a hashing bucket. As the space is broken down into discrete regions, the accuracy of values on the curve is given by the size of the smallest possible region, i.e., each value in the same region has the same Z-order hash. The size of the regions is determined by the number of bits and the size of the possible dimension values. To illustrate, Figure 2.6a shows a range of [0,15] describing each dimension, and a total of four bits for the Z-order curve. This means the range of each dimension is recursively halved twice. Each partition (one Z-order value) then represents a range of four values in each dimension ([0,4), [4,8), etc.). It is then clear that both point B and point C in Figure 2.6a are hashed to the same Z-order value, as they are both a part of the same X-partition [8, 12), and Y-partition [4,8). From this, each Z-order value represents a sub-range with a size given by:

$$\text{Range} \quad \text{Size} = \frac{(t-s)}{2^n} \tag{2.3}$$

Where $[s, t]$ describes the interval of possible values for a dimension and n, the number of bits for that dimension.

2.4.6 Locality Preservation

Figure 2.6a illustrates the localisation preserving characteristics of the Z-order curve. Most points close together in the two-dimensional space will also be close on the Z-order curve. However, there are some edge cases. Points C and E are close together in space, and their Z-order values (6 and 7) are also next to each other with no values in between. However, point D can also be said to be close in space to E, but on the Z-order curve, there are five values between them (8 to 12). For more bits and partitions, the distance on the Z-order curve grows even larger. For each split of the dimensions, such discrepancies will occur, with the first splits resulting in the largest jumps in values. As Figure 2.6a demonstrates, the biggest jump is the first horizontal split along the Y-axis. The worst case is then if two points are near the origin, but one point is in the northwest region, and the other is in the southeast region.

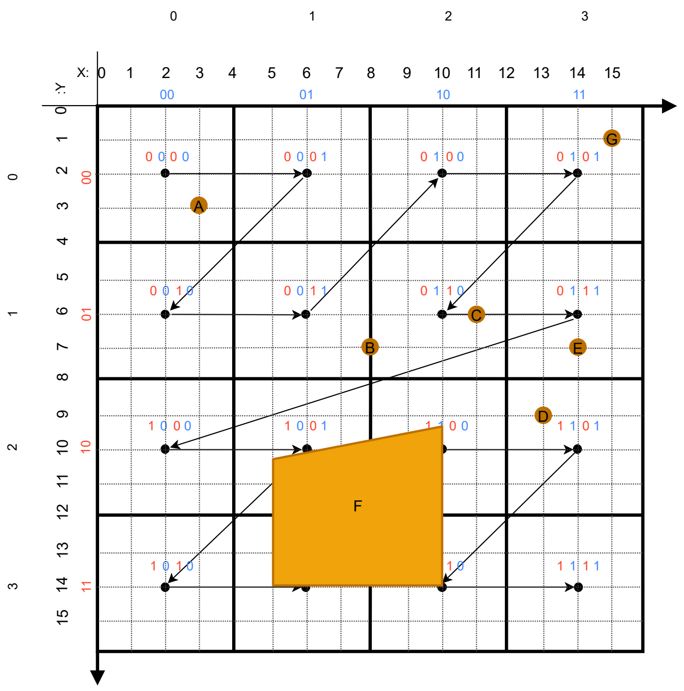
Related to the inverted index, if an index consists of Z-order values, 2-D points which are close in proximity in the 2-D space, are also generally close in the index (when sorted by indexing key). This means proximity searches using the Z-order values are more efficient as the points are kept close together in the main memory on disk.

2.4.7 Searching the Z-Order Curve

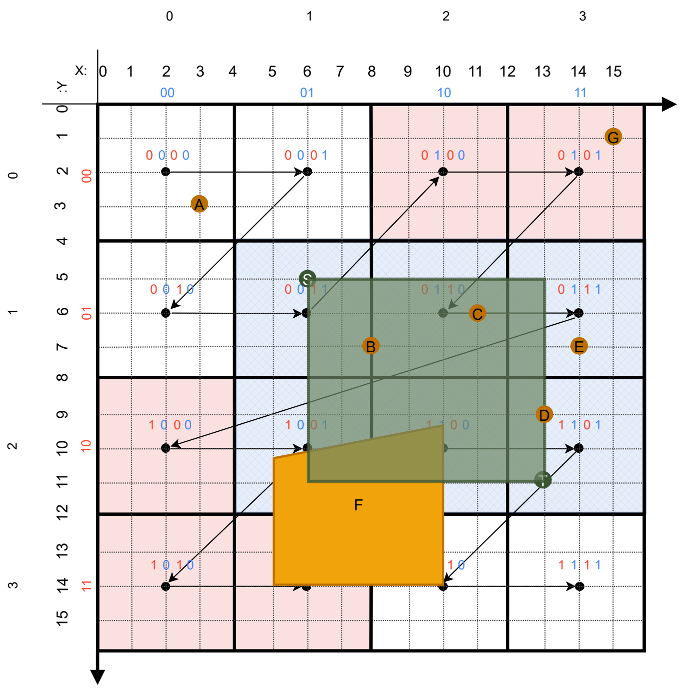
One benefit of the Z-order curve in indexing is that the hash values can be used in spatial queries without decoding them. As the Z-values are 1-dimensional, the spatial queries translate one-dimensional range searches over the values.

Naive Approach

The naive approach would be to do a one-dimensional range search over all the Z-order values, i.e., translate the lower and upper bound of the search into the equivalent Z-order values, and retrieve every Z-order value between these values. However, because of the edge cases present in the locality preservation, the values returned could include several values outside of the actual two-dimensional range. Figure 2.6b illustrates an MBR range search from point S (6,5) to point T (13,11), depicted by a green rectangle. The blue area depicts which Z-order values should be returned based on the range search. Point E will also be returned, as the Z-order regions of the search are larger than the actual search MBR and includes the region where point E is hashed. First, the two points are translated into their Z-order values. Using the naive approach, any point hashed to a Z-order value between the Z-order values of S (3) and T (13) is returned. As can be seen, this includes several non-relevant regions, depicted by the red rectangles. In this case, point G would also be returned even though it is outside of the search MBR. The search is thus more inclusive than necessary, and the resulting search precision is lower. Still, it does include all relevant regions, meaning the search recall is kept consistent. The Z-order value can therefore be utilised as a granular filtering



(a) Points and shapes can be represented by their Z-order value(s) in the 2-D plane.



(b) Illustration of how Z-order curves can be utilised in two-dimensional range search.

Figure 2.6

method. Lee et al. [18] utilise this characteristic for finding candidate matches of geographical queries.

BIGMIN and LITMAX

Tropf and Herzog [19] implemented a more sophisticated approach for range searches using the Z-order curve. The authors do this by iterating through the stored objects, sorted by the Z-order value, just as with the naive approach. However, when they detect an object which is outside of the search range (e.g., point G of Figure 2.6b, they initiate a calculation to find the largest previous Z-order value within the range (LITMAX) and the next smallest Z-order value within the range (BIGMIN). Any value between LITMAX and BIGMIN is then outside the search range, and the search can then continue in two sub-ranges: from the range minimum code to LITMAX, and from BIGMIN to the range maximum point. This is done recursively until all values of the range have been searched. The authors calculate these values by using a lookup table of 3-bit combinations, each bit being the MSB of the dividing, range minimum, and range maximum codes.

Using Figure 2.6b as an example, the records of the search from S to T (ignoring F) would be G, B, C, E, and D in that order. When reaching G, there is a split in the search area. BIGMIN is the next Z-order value inside the search range, larger than the current record value, in this case, 0110_2. LITMAX is the largest Z-order value inside the search area, but less than the value of the current record, in this case, 0011_2. The search is broken down into a search from the lower bound (S) to LITMAX, and from BIGMIN to the upper bound (T), i.e., the search [3, 13] is broken into searches [3,3] and [6, 13]. Each time a record with a Z-order value outside of the search range is encountered, new BIGMIN and LITMAX values are calculated, and the search is further split in two. No other values are outside of the index, so the search would stop when point D is reached.

2.4.8 Existing Systems

Even though the discovery of space-filling curves was done two centuries ago, the applications are more recent. The transformation of multidimensional data into 1-D representations is useful in indexing and searching as it makes it possible to keep using existing algorithms and data structures independent of dimensionality. When utilized in indexing the number of entries for a multidimensional data object can be reduced, and in turn, potentially reduce search execution time.

As mentioned, Tropf and Herzog [19] used Z-order curves for multi-dimensional range search in dynamically balanced trees. The experiments indicate a logarithmic time complexity with the number of records. Lawder and King [20] present different approaches of using space-filling curves for multi-dimensional indexing. This was done over a triple store, using a similar approach as BIGMIN and LITMAX, over the Hilbert Curve. GeoWave [21] utilised space-filling curves to reduce the dimensionality of geospatial data to index in key-value stores. MD-HBase [22] uses Z-order curves for multi-dimensional indexing over HBase.

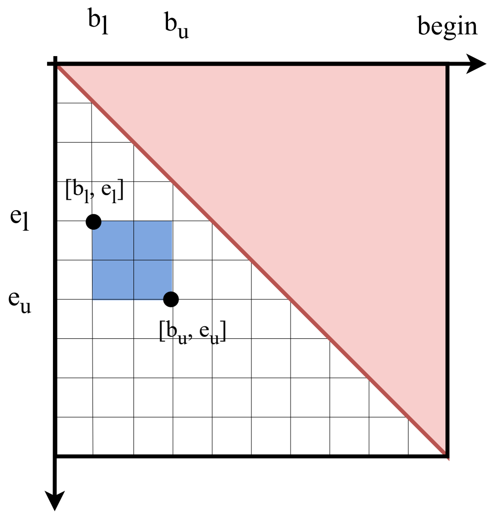


Figure 2.7: Temporal and numerical expressions can be modelled as a quadruple $T = (b_l, b_u, e_l, e_u)$ denoting the lower and upper bounds of the time interval beginning and end point values.

Haverkort and Walderveen [23] test different variants of the space-filling curve and their efficiency in organising the space into bounding-box hierarchies. Geohash.org [2] provides an API for encoding latitude, and longitude pairs into base32 Z-order values, called Geohashes. Similar approaches have been used in many open-source and commercial NoSQL database systems such as MongoDB [24] and Elasticsearch [25]. Lee et al. [18] used Geohashes to index spatial data in HBase, supporting fundamental spatial queries by utilising the hierarchical characteristics determined by the length of the Geohash.

2.5 Uncertainty-Aware Model

Temporal expressions such as “in the 1930s” have an inherent uncertainty. The phrase does not specify when exactly in the 1930s it is taking place and could be any yearly values in the interval [1930, 1939]. The same holds true for numerical expressions. “400 million USD” can represent any value in the interval [400M-499M]. This can be modelled by imposing lower and upper bounds of the beginning and end of the intervals. The uncertainty-aware time model [26] can be used to represent temporal expression with inherent uncertainty present. The temporal expressions are part of a discrete time domain with timestamps $t \in \mathbb{Z}$, each timestamp describing the number of time units since a reference time point. The temporal unit is the lowest discrete point in the model, such as seconds, days, or years. The temporal expression is modelled as a quadruple,

$$T = (b_l, b_u, e_l, e_u), \tag{2.4}$$

where b_l and b_u denote the respectively lower and upper bound of the beginning boundary of the time interval. Similarly, e_l and e_u denote the lower and upper bound of the end boundary of the time interval. Any expression can then be modelled as a time interval $[b, e]$ having $b \in [b_l, b_u]$ and $e \in [e_l, e_u]$, with the constraint that $b \leq e$. Figure 2.7 illustrates this time model. The blue square shows the range of possible values represented by a temporal expression. The red area illustrates the area outside of the constraints of the time model. As is evident from the illustration, the time model represents a minimum bounding rectangle, or square, where one corner is given by the lower bounds for each dimension, and the other corner is given by the upper bounds. For example, "the 1930s" translates to the quadruple $[1930, 1939, 1930, 1939]$, being modelled as the time interval $[1930, 1939]$. This can be represented by the MBR with representative corners (1930, 1930) and (1939, 1939). The MBR then spans any time interval following the constraints of the time model. The same holds true for any numerical range with the same restrictions.

Using temporal information in indexing has been done in works such as [27, 28]. In [27], the authors index timestamp versioned web-based document collections. They utilize a partition-based approach to split posting lists corresponding to different timestamps enabling quick execution of timepoint-based queries. Anand et al. [28] build on this by proposing a query optimization framework able to query time intervals with I/O constraint.

Chapter 3

NASH

This chapter describes how the annotated documents are modelled in a system to support full-text semantic search with range-search capabilities over two-dimensional annotations. The system is called NASH, derived from *Numerical* and *Hash*, signifying the core concepts of the system. The query operations the system supports, and how the index is designed to support these operations efficiently are described. Then, the hashing implementation using Z-order curves is explained in detail with related range search optimizations.

3.1 Data Model

This section describes the data model used for indexing document collections and their annotations. NASH builds on the annotated text model used by Gupta and Berberich [9, 10]. The documents can be represented by a layered model of text and annotations as seen in Figure 3.1. Each document d in a collection $\mathcal{D}$ is modelled as a collection of annotation layers $d_{\mathcal{L}}$:

$$d_{\mathcal{L}} = \langle l_{[i,j]} \dots l_{[p,q]} \rangle \tag{3.1}$$

3.1.1 One-dimensional Values

Each annotation l is one of the entity types available in the entity type vocabulary $\sum_{\mathcal{L}}$. NASH uses the following types {PERSON, ORGANIZATION, LOCATION, TIME, NUMBER}, and corresponds to the course-grained annotation types extracted from NLP entity extraction. Each of the annotation layers above shares this positional information as well: $l_{[i,j]}$ decorates a contiguous word sequence $\langle w_{[i,j]}, \dots w_{[p,q]} \rangle$ in the word layer $d_{\mathcal{W}}$. The word layer can be treated as an annotation layer as well, consisting of words drawn from the vocabulary of the entire document collection $\sum_{\mathcal{V}}$. The entity layer describes the entity types of word tokens if they have one. For example, the words "Alan Turing" is in positions 0 and 1, and the entity type layer has a PERSON token in the same positions. The named entity layer represents

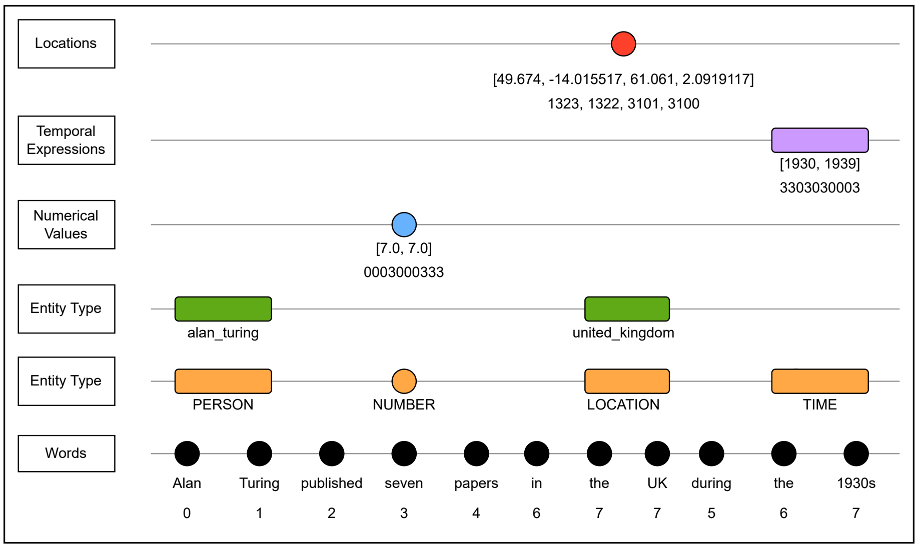


Figure 3.1: The data model representing the document text and its annotations. The model is inspired by the work of Gupta and Berberich [9, 10].

a normalized token of the named entities, such that if an entity is described with different words, it still has the same normalized token in the named entity layer. The layer thus describes unique instances of named entities.

3.1.2 Two-dimensional Values

For numerical and temporal values, the textual representation is converted into respectively its number- and time-value intervals. As the figure shows, “the 1930s” is represented by a ten-year temporal range, and "seven" is represented by a zero-length interval. Furthermore, “the 1930s” translates to the quadruple [1930, 1939, 1930, 1939] when using the time model described in Section 3.1 with the year zero as a reference point and a one-year temporal unit. For positional phrase queries, only a year-level precision is utilised for simplicity and efficiency. For any extracted temporal expression, $b_l = e_l$ and $b_u = e_u$. Thus, the expression can be modelled as a single spatial point in a two-dimensional coordinate system where the x-axis represents the lower bound of the interval, and the y-axis represents the upper bound. The phrase “the 1930s” is then represented as the point (1930, 1939) in the year temporal coordinate system. The same logic is true for the numerical annotations, but with different levels of precision (described in Section 4.3.1). As these intervals are represented by a single point, they can be translated to a single Z-order hash, which preserves the spatial properties of the annotations. These hashes are shown beneath the numerical/temporal values in Figure 3.1.

The geographical modelling itself is of secondary concern but is included to show how applicable the hashing approach is for different annotation types. A simplified Minimum Bounding Rectangle (MBR) approach is used to represent geographical areas. As locations are represented by an MBR and not single points, it cannot (always) be reduced to a single Z-order hash and still be precise. A process analogous to a Z-order range search (described in Section 3.4) has to be done on the MBR to calculate the minimum Z-order hashes which cover the area, which are all stored at the location layer.

The key contribution of the data model is that the annotation layers keep the positional information of the word tokens, they describe. As the positional information between the layers is kept consistent, queries can be formulated by combining tokens from all layers as part of a single query.

3.2 Query Operators

A set of query operators is needed to support full-text semantic search over the annotated document collection. This includes searching with positional spans for tokens and annotations, within a text region. The smallest region is in this case sentences. This means that matches are only found within the tokens of a sentence or spanning multiple sentences. A search over the index is a function with the query Q as input, and a posting list of documents $\{d_1, d_2, ..., d_k\}$ matching the query as output.

3.2.1 Text Tokens and One-Dimensional Annotations

The **Boolean operators** (Section 2.1.4) are used to search over tokens and annotations, as well as combining sub-queries. The AND ($\wedge$) operator is used to match all terms in the query. For example, for word tokens this can be a query such as *david*$\wedge$ *hilbert*$\wedge$ *wrote* $\wedge$ *a* $\wedge$ *paper* to match a text phrase. A text phrase is a sequence of tokens which are one after the other, with consecutive positional spans. To simplify the notation, this can be combined into a single bracket: *[david hilbert wrote a paper]*. A text phrase can also span across multiple layers, e.g., the annotation layer can be used in a query such as [*(PERSON) wrote a paper*], being more general than the former query.

To combine annotations with text tokens, a **Stack Operator** [9] is used. Annotations from an annotation layer can be stacked on top of the text layer tokens, such that the positional spans of the text token and the annotation token are the same. The *STACK*($\oplus$) operator represents this operation. For example, to search for a specific text phrase annotated as a person, a query such as $(PERSON)\oplus$ *[david hilbert]* can be constructed. Brackets indicate a text phrase (one or more tokens next to each other), and parenthesis indicates an annotation. Text tokens can have multiple annotations, and especially the location annotations can have multiple associated Geohashes in the location annotation layer.

The **Span Operator**(δ)[1] supports queries with wildcard tokens between the query terms. The span operator determines how many tokens (words/annotations) can be between the query terms, and it still is a match in the document.

[1]The naming is inspired by the span query of Elasticsearch/Lucene, with similar semantics.

The operator is associated with an integer number, the *slop*, indicating the number of tokens the matching text phrase can span. A slop of zero then translates to a phrase query, i.e., the tokens have to be next to each other and in order. A negative number indicates that the match can span the entire document. Additionally, the span operator can be used to indicate if the match should be within a sentence, or between a specified maximum number of sentences. For example, to find matches within a sentence context, a query such as $\delta\langle$ *[david hilbert] [wrote paper] (TIME)* $\rangle$ can be constructed. The angled brackets indicate that the slop is between every phrase (square bracket) and tokens/annotations within the brackets. This query matches any text regions in documents where *david hilbert*, *wrote paper*, and a text phrase annotated as TIME is within the same sentence.

3.2.2 Range Queries

To support spatial range-based queries (see Section 2.3.3 over numerical, temporal, and location annotations, the **Range Operator**($\Delta[b_l, b_u, e_l, e_u](a)$) is used. This operator is always connected to one of the applicable annotation types, a: *TIME*, *NUMBER*, or *LOCATION*, and a search MBR. The operator allows for matching with indexed hash tokens of the same annotation type, which are spatially within a certain two-dimensional range. The operator thus defines a range with lower bounds (l) and upper bounds (u), for the begin (b) and end (e) of the interval, equalling a search MBR. Any sub-interval within the search range is found as a hit. For example, the query $\langle$ Δ*[1885, 1885, 1895, 1895](TIME)* $\rangle$ matches text regions with temporal expressions between 1885 to 1895. The three types of range-based operations defined in Section 2.3.3 can be implemented using range queries:

$$\text{contain}(l) = \{d \in \mathcal{D} | l' \in d_{\mathcal{L}} \wedge l' \cap l = l'\}, \tag{3.2}$$

$$\text{intersect}(l) = \{d \in \mathcal{D} | l' \in d_{\mathcal{L}} \wedge l' \cap l \neq \emptyset\}, \tag{3.3}$$

$$\text{proximity}(l) = \{d \in \mathcal{D} | l' \in d_{\mathcal{L}} \wedge l' \cap l = \Delta\}, \tag{3.4}$$

where Δ is a a proximity interval defined by the range operator.

3.2.3 Range Search MBR Construction

The approach used for calculating search MBRs differs slightly between the temporal and numerical hashes compared to the location hashes. The indexed location annotations are represented by a polygon and can thus have multiple hashes associated with it of varying lengths. Comparably, the numerical and temporal annotations are represented as single points. The range-based queries are thus more expensive for location annotations as more hashes are computed and must be searched. Spatial queries over geographical Z-order hashes are also explored in previous works [18, 21]. The rest of this thesis focuses on the numerical and temporal hashes only. Figure 3.2 visualises the MBRs generated for the temporal

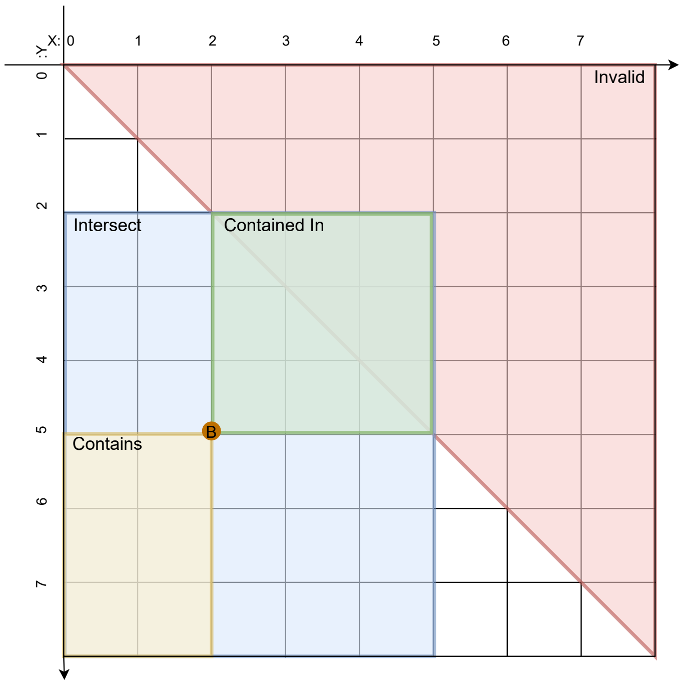


Figure 3.2: An example of how the generated MBRs for the search range B ([2, 5], [2,5]) can be visualised for numerical and temporal annotations.

and numerical annotations for each type of range-based query. The illustrative search/index range is illustrated by point B, where $X \in [2, 5]$ and $Y \in [2, 5]$, over the maximum range $X, Y \in [0, 7]$. The MBR construction is defined as follows:

Contains The contains query searches for the two-dimensional range where $X \in [b_x, e_x]$ and $Y \in [b_y, e_y]$, with the additional restriction, that $X \leq Y$. The MBR has corners with coordinates (b_x, b_y) in the northwest corner and (e_x, e_y) in the southeast corner. Any indexed point (representing a range) where $b_x \leq X \leq e_x$ and $b_y \leq Y \leq e_y$ matches the query. Figure 3.2 illustrates the contains MBR as the green area with the MBR (2,2), (5,5). Any point to the northeast of the indexed point will have a larger X (begin) value and a smaller Y (end) value than the range [2,5]. As can be seen, the range is likely to have multiple hashes in the invalid area which can be ignored by the range search. Intuitively this makes sense as points near the diagonal line will have X and Y values close together, indicating a smaller range.

ContainedBy The containedBy query is used to find the indexed annotations that cover the search range. As all indexed ranges (points) have the same lower and upper bounds for both dimensions, i.e., $b_x = b_y$ and $e_x = e_y$, the search MBR needs to have the same restriction. The search range can be represented by the range $[b, e]$, indicating shared bounds between the two dimensions. The containedBy query searches the two-dimensional range where

$X \leq b$ and $Y \geq e$. The lower bound for X and the upper bound for Y are the lower and the upper bound of the Z-order curve, respectively. Figure 3.2 illustrates the containedBy MBR with the yellow shaded area for the search range [2,5]. Any indexed range which contains the range [2,5] will be in the southwest direction as any point to the southwest of the search range, point B, will have a large Y value and a smaller X value than B.

Intersect The intersect query matches any indexed range which overlaps with the search range. Again, using the fact that indexed ranges are bound by $X, Y \in [b, e]$, the intersect query equals a range search with restrictions $X \leq e$ and $Y \geq b$. Similarly, the lower and upper bounds are limited by the maximum Z-order ranges. Figure 3.2 shows the resulting search MBR for the range [2,5] as the blue area. The intersect query is thus the most inclusive, including all values from both the contains, and containedBy queries, in addition to those that overlap the search range partly.

Proximity Proximity between hashes can be calculated in different ways. Here, the proximity is based on the Euclidean distance in space, i.e., the shortest possible distance between indexed ranges. With the query point given by $[b, e]$, and the distance Δ, the proximity MBR equals point $(b - \Delta, e - \Delta)$ in the northwest corner and point $(b + \Delta, e + \Delta)$ in the southeast corner. For example, finding points within a distance of one of B (range of [2,5]) in Figure 3.2 creates a search MBR with corners (1,4) and (3,6). For simplicity, the distance is uniform in every direction but could differ based on the proximity needs. For example, one could search for events near a date with more results from recent years by having a greater distance in the Y-direction (end) compared to the X-direction (begin).

For smaller distances, the proximity search can be more restrictive than the other range-based queries. This can be a benefit to narrow down the number of hits. Compared to the intersect query, the results can be a combination of both containedBy and contains matches but can also match with intervals not overlapping but which are still nearby.

The range-based queries can also be combined with each other simply by simply adding to or subtracting from the resulting MBR hashes. For example, to find intersect results without any ranges being fully contained by the other, the hashes for contains and containedBy can be calculated and subtracted from the intersect hashes. As all the hashes can be calculated at the client-side, no extra computation is needed when searching the index.

3.3 Index Design

To support full-text semantic search, inverted indexes are utilised. This allows for scalability and reliability, as well as compatibility with many existing databases and storage solutions. As all two-dimensional annotations are linearized to a single dimension, no modifications to the inverted index are needed. Each of the layers in the data model can be seen as an inverted index, but the positional information is kept consistent between them.

3.3.1 Combining Words and Annotations

Using an inverted index over the text field, and storing the document id and term positions along with the terms enables searching for both single words and phrases of multiple words. Annotations are stored similarly in their own inverted indexes, sharing the same token positions as the words they describe. An annotation spanning more than one-word token will share the same start token position and have a token length attribute equal to the number of words it describes. Figure 3.3 displays a conceptual model of the index and how the positions are stored. For example, *David Hilbert* can be indexed as two tokens, *David*, and *Hilbert*, with positions 0 and 1. The two tokens are together marked as a *PERSON* entity type, stored as a separate term at position 0. Similarly, the combined entity instance *david_hilbert* is also indexed at position 0. In the same manner, the token *1891* at position 6 has an associated entity type *TIME*, and the temporal hash 3300030030_4[2] at the same position. This representation is equal for all the other layers not shown. The sharing of index positions allows for the stack operator to efficiently be executed by restricting tokens and their annotations to be at the same position in the search.

Term	Posting List
PERSON	[docID=0, position=0, positionLength=2], ...
david&hilbert	[docID=0, position=0, positionLength=2], ...
david	[docID=0, position=0], ...
hilbert	[docID=0, position=1], ...
published	[docID=0, position=2], ...
a	[docID=0, position=3], ...
paper	[docID=0, position=4], ...
in	[docID=0, position=5], ...
TIME	[docID=0, position=6], ...
1891	[docID=0, position=6], ...
3300030030	[docID=0, position=6], ...

DOC0:	David	Hilbert	published	a	paper	in	1891
	0	1	2	3	4	5	6

Figure 3.3: A model of the index for annotations and text.

[2]Indexed annotations for temporal and numerical values have equal lower and upper bounds for both dimensions, resulting in hashes consisting mostly of 0s (00) and 3s (11) in a base 4 encoding.

3.3.2 Sentence Regions

Having the positional information stored in the index allows for extracting relevant text regions matching the search, i.e., that the last matching token should be at most *n* positions away from the first matching token. NASH also support a sentence-level region search, such that the matches are kept within single sentences. This is achieved by using a positional jump of 10,000 values between the end of one sentence to the beginning of the next. To only allow matches within a sentence context, the maximum position gap of the span operator is set to 10,000. No tokens from different sentences can be within 10,000 positions of each other, and they can thus never match in the same region. Using a 20,000 search-span extends this to a two-sentence search region, etc.

3.3.3 Z-Order Hashes

To be able to utilise the hierarchical properties of the Z-order hashes and the range query operator, prefix searches are used. To make this kind of search more efficient, each of the hash annotations also stores its prefixes as part of the inverted index. For example, for the hash 3300030030 corresponding to the range [1890, 1890], each prefix 330003003, 33000300, 3300030, etc. is indexed at the same position.

Z-Order Hash Calculation

Algorithm 1 shows the procedure of the calculation of Z-order hashes. The equivalent Java code implementation can be found in Appendix A.1.1. The procedure is the same as described in Section 2.4.4. However, the bits are calculated directly one at a time instead of interleaving them at the end. The algorithm takes an X, and Y coordinate pair, the max ranges for each dimension, and the desired precision as input. The Z-order value is calculated one bit at a time, starting with the MSB. This represents the first Y-value of the hash. A counter, (*b*), is used to indicate the current bit position. If the position is even, the current bit represents a part of the Y-value, and if not, it represents a part of the X-value. The calculation of the current bit is the same for both dimensions. The mid-point of the current dimension is calculated and compared against the mid-point of the current range for that dimension. It starts as the X and Y dimension bounds and is halved for each bit being calculated. If the value is larger than or equal to the midpoint, the current bit position is set to 1, and if not, it stays at zero. The range of the current dimension is then halved, keeping the half containing the value. This is done alternately for the X and Y dimension, until the desired number of bits, or precision, is reached.

Decoding of the hash is done in a similar manner, but in reverse. Starting with the MSB, if the bit is a 1, the maximum range of the dimension (starting with Y) is split into the upper half, and if the bit is 0, into the lower half. This is done for each bit, each time halving the range for the current dimension looked at. The

Algorithm 1 Calculate the Z-order value/hash of a point given by the X and Y coordinates.

Input: X-coordinate x, Y-coordinate y, Range of X XR, Range of Y YR, Precision p

Output: h: the calculated Z-order value

```
 1: procedure CALCULATEHASH($x, y, XR, YR, p$)
 2:     $h \leftarrow$ bit string with zero-bits, length of $p$        ▷ 00...
 3:     $b \leftarrow 0$                                              ▷ Current bit position
 4:     while $b < p$ do
 5:         if $b$ is even then                                      ▷ Y-dimension
 6:             $mid \leftarrow (YR[0] + YR[1])/2$
 7:             if $Y \geq mid$ then
 8:                 $YR[0] = mid$
 9:                 $h \leftarrow$ set current bit of $h$ to 1
10:             else
11:                 $YR[1] = mid$
12:         else                                                     ▷ X-dimension
13:             $mid \leftarrow (XR[0] + XR[1])/2$
14:             if $X \geq mid$ then
15:                 $XR[0] = mid$
16:                 $h \leftarrow$ set current bit of $h$ to 1
17:             else
18:                 $XR[1] = mid$
19:         // next bit
20:         $b \leftarrow b + 1$
21:         $h \leftarrow$ left shift h by 1
22:     return $h$
```

result is a range for each dimension, indicating the region the hash covers, with the size of the range given by Equation 2.3.

As the range search over the Z-order curve is at minimum done at two bits at a time, a base 4 encoding is used. This means each base 4 character represents one partition in both the X and the Y dimensions. This results in a shorter textual representation while still allowing for the finest precision reduction or increment by adding or moving to the hash. The encoding is as simple as translating each X, and Y pair into its base 10 representation. $00_2 \rightarrow 0_{10}, 01_2 \rightarrow 1_{10}, 10_2 \rightarrow 2_{10}, 11_2 \rightarrow 3_{10}$. As two and two bits are encoded together, this results in a base 4 encoding.

3.4 Z-Order Range Search

The range search over the Z-order curve is done by recursively splitting the search range into Z-order ranges which are continuous at the current recursion level (recursion level, tree depth, and bit-length/precision can be treated as describing the same concept). This is done by utilising a modified version of the previously mentioned *BIGMIN* and *LITMAX* approach described by Tropf and Herzog [19] (see Section 2.4.7). Their approach iterates through indexed Z-order objects and initiates a sub-range search if an object is found to be outside of the current search range. This means the calculation is done as part of the query processing, and only when an invalid value is encountered. The approach used in NASH is to compute the relevant Z-order values before the search, which allows for normal term-based matching in the index. However, to keep the search from enumerating all the hashes of a search range at the finest possible precision, three methods are combined to reduce the number of hashes and their length. The concept of continuous Z-order values is used to stop the search at a given depth, resulting in shorter hashes and a minimum set of Z-order hashes which cover the search region. Additionally, a method for searching ranges at a lower accuracy is implemented which calculates the minimum Z-order precision given a maximum allowed range deviation. Lastly, the boundary restrictions of temporal and numerical intervals ($X \leq Y$) are used to stop the search if it moves into an invalid area.

3.4.1 Optimizations

Continuous Z-Order Values

The first method to reduce the number of search hashes is to check if the lower and upper Z-order values of the search are continuous. Two Z-order values being continuous is defined as every Z-order value laying between the interval defined by the two values, are inside the search region. E.g, in Figure 3.4a, the hash values for S and T are not continuous, as there are four Z-order values (at the current precision level) between them which are outside of the search range. Region 25 to 28 on the other hand is continuous, as travelling through them in order does not produce a value outside of the range [25, 28].

The bit-characteristics of Z-order values can be used to check if two hashes are continuous. First, the number of common MSBs between the two hashes is calculated. This can be done by a bitwise XOR operation between the hashes and counting the number of leading zeroes. The number of common MSBs indicates the current precision level (or depth in the tree) that is being searched. For example, zero common bits mean that the two hashes are in different quadrants at the lowest precision level. Then, the remaining bits after the common bits are checked. The Z-order value of the lower bound must have all zeroes after the common bits, indicating that it is the first quadrant of the sub-range being investigated. The upper bound must have all one-bits following the common bits, indicating that it is the last quadrant of the same sub-range. If this is the case, the two hashes are continuous, and only the common MSBs are needed to represent that region. As all Z-order computations in this thesis are done at least two bits at a time (both X and Y), a continuous Z-order interval will necessarily travel through all four quadrants of a region, meaning that they can be represented by the parent region. For example, the range from 25 to 28 in Figure 3.4b can be represented by the single value L as the quadrants cover the entire region, resulting in a shorter hash.

Range Search Accuracy

Another way to make the range search more efficient is to reduce the precision of the curve as much as possible while still being within a minimum range accuracy. The accuracy of a single dimension on the curve is given by Equation 2.3. Solving this equation for n and rounding up to the nearest integer gives the minimum number of bits needed to be certain that any hashes calculated are not deviating more than the allowed maximum for that dimension. This results in Equation 3.5. As the range search is done over two dimensions, the calculated precision must be multiplied by two. If the dimensions differ in required accuracy, the precision required for the most accurate of the dimensions can be used. The calculated precision (hash length) is only the maximum precision needed and can be lowered for continuous hashes.

$$\frac{(t-s)}{2^n} \leq \text{Deviation} \tag{3.5}$$

For larger ranges, potentially more hashes will be returned from the Z-order curve, as the MBR is more likely to cover multiple quadrants of different levels. At the same time, searches over larger areas often naturally require less precision. For example, if searching for temporal annotations in the range [1900, 1900] to [1999, 1999], having a potential ±10 year deviation in each end is not of as great concern, compared to if the search was from [1990, 1990] to [1999, 1999]. This is exploited by defining the maximum allowed deviation as a percentage of the input search range. A larger search range allows for a greater deviation, and a smaller search range requires a smaller deviation. For example, by multiplying the smallest of the input ranges by 0.1, a 10% deviation is achieved in the worst case. The larger range [1900, 1900] to [1999, 1999] would then potentially allow for a ± 10 year accuracy, while the smaller range [1990, 1990] to [1999, 1999] results in a ± 1 year accuracy.

Pruning Invalid Hashes

For temporal and numerical expressions, the restrictive characteristics of their lower and upper bounds can be utilised to reduce the number of resulting Z-order hashes from the search. Recalling from Section 2.5, there are constraints on which points in the coordinate system are allowed for the time model, and the same holds true for the numerical values. Only points where $X \leq Y$ are valid, indicated by the non-shaded area in Figure 2.7. This fact can be utilised in the Z-order range search by ignoring any sub-range which fully takes place in the invalid area. The initial search starts with both hashes for the MBR outside of the invalid area, but when the MBR is split into new sub-ranges to be searched, some sub-ranges may be entirely within this area and the sub-range search can safely be stopped. As the northwest corner of the MBR is inside the invalid area, so is the southeast corner and thus the entire MBR, meaning only the northwest value must be checked.

The values of the invalid area follow a recursive pattern. At the topmost level, the entire northeast quadrant can be ignored along with two smaller triangles. These triangles can recursively be broken down into smaller quadrants until the hashes covering the entire invalid area are found. The topmost northeast quadrant is given by the hash 01_2. Any value having this as its prefix is then invalid and can be ignored. For example, in Figure 2.3a, the entire region C can be ignored. The same is done for the two remaining triangles. Moving one level down in the southeast quadrant, D, the hash for the northeast region O is 1101_2. Similarly, moving down in level in region B yields a northeast hash of 0001_2. Each invalid hash can be calculated in this way, by individually adding 00_2 (northwest quadrant) and 11_2 (southeast quadrant) to the end of all ignored values of the previous level. The number of ignored values at each level thus grows at a rate of 2^n, where n is the current level, starting at 0. The total number of ignored values for a given level of precision is then $2^n + 2^{n-1} + 2^{n-2} + ... + 2^0 = \sum_{i=0}^{n} 2^{n-i} = 2^{1+n} - 1$. These values are stored in a lookup table. As part of the range search of temporal and numerical hashes, an additional check is done by seeing if the northwest hash of the current sub-range is in this table, meaning the search in the current region can be stopped and ignored, resulting in fewer hashes.

3.4.2 Range Search Algorithm

Combining the described optimizations with the modified BIGMIN LITMAX approach yields the Z-order range search algorithm (1). The equivalent Java code can be found in Appendix A.1.2. The input of the search consists of the two hashes corresponding to the corners of the MBR covering the search area, a list of hashes calculated so far (starting with none), the set of the starting maximum range for each dimension, and the maximum allowed precision of the curve. The input maximum precision can be calculated using the previously described deviation (Equation 3.5) approach. Using the Z-order model as in Figure 2.6b, this translates to a search from the northwest to the southeast corner of the MBR. The calculation of BIGMIN and LITMAX is used to skip any hashes outside of the search area. These calculated values result in a split of the search over two new sub-ranges. The algorithm is thus recursive as it breaks the range search into smaller and smaller sub-ranges, like moving down levels of a quad-tree. The traversal stops when one of the stop-conditions is reached, either that the current sub-range is continuous, or that the maximum precision is reached.

Algorithm 2 Calculates the minimum set of Z-order hashes covering a search area

```
Input: Northwest MBR point NW, Southeast MBR point SE, List of hashes H (initially
    empty), and maximum Precision p
Output: A minimum set of hashes H covering the input MBR
 1: procedure RANGESEARCH($NW, SE, H, p$)
 2:     $cb \leftarrow$ number of common MSBs between NW and SE
 3:     if $NW$ and $SE$ are continuous then                    ▷ range within MBR
 4:         $h \leftarrow$ bitstring equal to the common MSBs
 5:         $H.add(h)$                                          ▷ hash shorter than $p$
 6:         return $H$
 7:     if $cb = p$ then                                        ▷ at lowest level
 8:         $H.add(NW)$                          ▷ Single partition, max hash length
 9:         return $H$
10:     // Calculate BIGMIN $B$ and LITMAX $L$
11:     if $cb$ is even then
12:         $B, L \leftarrow calculateBigminLitmax([NW.X, SE.X], [NW.Y, SE.Y], 1)$
13:     else
14:         $B, L \leftarrow calculateBigminLitmax([NW.Y, SE.Y], [NW.X, SE.X], 0)$
15:     $rangeSearch(NW, L, H)$                                 ▷ West/North split
16:     $rangeSearch(B, SE, H)$                                 ▷ East/South split
17:     return $H$
```

The algorithm starts by finding the number of common MSBs (line 2). This is used to check if the NW and SE hashes are continuous. If they are continuous, the search can stop, and the common MSBs can be added to the return hash set (line 5). For example, searching from point C to point E in Figure 2.6a at a four-bit precision means no splits of the search are necessary. C = 0110_2 and E = 0111_2, i.e., they have three common MSBs, 011_2. The next bit(s) for C is a single 0, and for E is a single 1, indicating that the two values are continuous. However, if the bit-precision of the curve was increased, they would no longer be continuous as their hashes would no longer have all zeroes or ones. The defined maximum deviation and continuous values thus function together to limit the number of hashes. If the number of common bits is equal to the maximum bit-precision, the search stops at the current recursion as the two hashes are seen as equal for the given precision, and the common bits are added to the return hashes (line 9). If none of the two stop conditions are true, the BIGMIN and LITMAX calculation must be done to find the X and Y values for splitting the range search (lines 10-14). Based on the calculated values, the search is split into two sub-ranges with MBR corners (NW, LITMAX) and (BIGMIN, SE).

Calculating BIGMIN and LITMAX

The BIGMIN LITMAX algorithm (1) calculates the two new MBR corners of the split (the old northwest and southeast corners are kept). The equivalent Java code can be found in Appendix A.2. The split is either horizontal or vertical, which is indicated by the number of common bits. An even number indicates that the next bit not in common is a Y-value (horizontal split), and an odd number indicates that the next bit is an X-value (vertical split). This means that either the X-values or the Y-values of the new corners are already known. The algorithm calculates the values of the unknown dimension and combines them with the values of the known dimension into the new search corners. A horizontal split means that the search is split along the Y-axis and that the current X-values are within the search range and part of the new corners. The BIGMIN, LITMAX calculation is then done to find the Y-values of the regions just north, and south of this line. Similarly, a vertical split means a split along the X-axis, and the calculation is done to find the X-values of the regions just to the left and to the right of the line.

The algorithm takes as input both the values of the known dimension as one pair and the values of the unknown dimension as another. For example., a horizontal split takes as input the X-bits for both the corners as the known values and the Y-bits for both the corners as the unknown values. Additionally, the algorithm takes as input a value indicating which dimension the calculations are to be done for (the unknown values), 1 indicating the Y-dimension. The algorithm outputs the LITMAX and BIGMIN values which respectively correlates with the southeast corner of the lower sub-range, and the northwest corner of the upper sub-range.

The calculation itself is done by simply appending a set of pre-defined bits (masks) to the end of the common MSBs of the unknown values (lines 3-6). First, the common MSBs of the unknown values are found. The common bits indicate at which precision level the dividing line of the split is. The BIGMIN mask is calculated by adding a 1 followed by zeroes ($1000..._2$), until the maximum bit-precision is reached. The 1 indicates the south/east (BIG) side of the split, and the zeroes indicate the minimum (MIN) of these values. Similarly, the LITMAX mask is calculated by appending a 0 followed by ones (0111...) until the maximum bit-precision is reached. The 0 indicates the north/west (LIT) side of the split, and the ones indicate the maximum (MAX) of these values. The final LITMAX and BIGMIN values are calculated by interleaving the values from the known dimension with the calculated LITMAX and BIGMIN masks for the unknown dimension (lines 7-12). LITMAX is interleaved with the largest of the known values, and BIG-

MIN with the smallest. The starting value of the final hash is always the Y-value, and the dimension indicator shows which dimension is the unknown and which is known to get the correct bit in the MSB position (line 7).

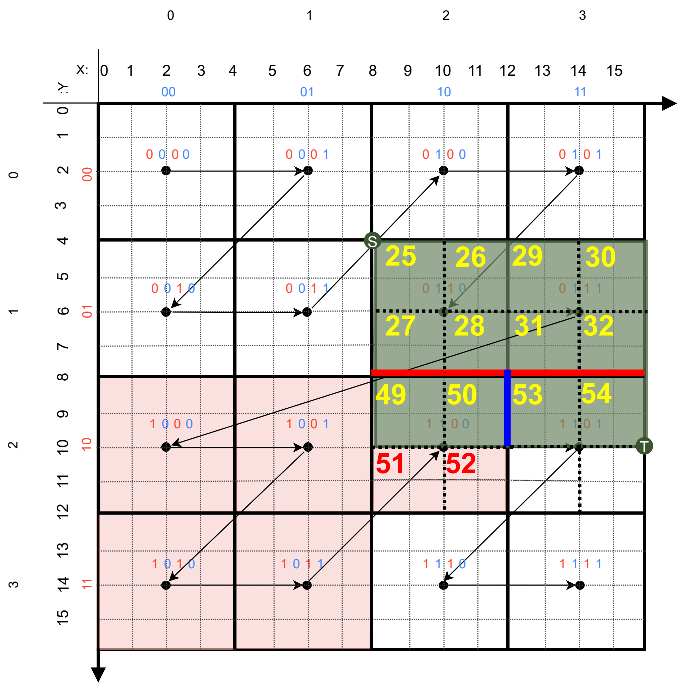
Algorithm 3 Calculates BIGMIN and LITMAX values

```
Input: Tuple of values for known dimension $K$, Tuple of values for unknown dimension
    $U$, Boolean indicator of unknown dimension $d$
Output: Calculated values for unknown dimension $litMax$ and $bigMin$
 1: procedure CALCULATEBIGMINLITMAX($K, U, d$)
 2:     $cb \leftarrow$ number of common MSB between $U[0]$ and $U[1]$
 3:     $litMask \leftarrow$ bit-string equal to 011..., total length equal $cb$
 4:     $bigMask \leftarrow$ bit-string equal to 100..., total length equal $cb$
 5:     $litMask \leftarrow$ first $cb$ MSBs of $U[0]$ followed by $litMask$
 6:     $bigMask \leftarrow$ first $cb$ MSBs of $U[1]$ followed by $bigMask$
 7:     if $d = 0$ then                                  ▷ Unknown dimension is Y
 8:         $litMax \leftarrow$ bit interleave $litMask$ and $K[1]$, starting with mask
 9:         $bigMin \leftarrow$ bit interleave $bigMask$ and $K[0]$, starting with mask
10:     else                                             ▷ Unknown dimension is X
11:         $litMax \leftarrow$ bit interleave $K[1]$ and $litMask$, starting with K
12:         $bigMin \leftarrow$ bit interleave $K[0]$ and $bigMask$, starting with K
13:     return $litMax, bigMin$
```

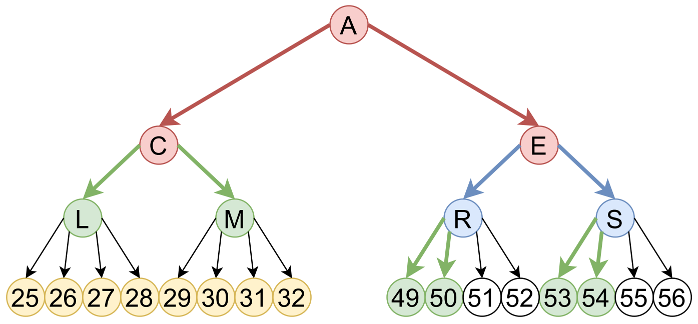
Range Search Example

Figure 3.4a illustrates how the range search is divided into smaller sub-ranges. The two-dimensional range search is done from point S to point T equals a Z-order range search from the hash in quadrant 25 (011000_2) to the hash of quadrant 54 (110101_2), over a 6-bit Z-order curve (only the four-bit hashes are shown). S and T are not continuous, as shown by the quadrants shaded red. The algorithm finds the number of common bits, which is zero, meaning a horizontal split and unknown Y-values. The extracted Y-bits from quadrant 25's hash is 010_2 and from quadrant 54 100_2. As the current Y-values share no common bits, the Y-bits for BIGMIN (the BIGMIN mask) is simply 100_2, and for LITMAX, 011_2. Interleaving 100_2 with the (known) X-bits of quadrant 25 (100_2) results in a BIGMIN of 110000_2, which is in quadrant 49. Similarly, interleaving 011_2 with the X-values of quadrant 54 (111_2) yields a LITMAX of 011111_2, quadrant 32. Based on these values, the search range is split into two sub-ranges shown by the red line in Figure 3.4a. The new ranges are quadrant 25 to quadrant 32, and quadrant 49 to quadrant 54.

The search from quadrant 25 (01100_2) to quadrant 32 (011111_21) is done in the same manner. However, this time the range is continuous. The hashes share three common bits, with the northwest hash having all zeroes following the common bits, and the southeast hash having all ones following the common bits.



(a) The Z-order range search can be visualised over a two-dimensional coordinate system.



(b) Tree representation of the Z-order range search

Figure 3.4: Range search over the Z-order curve done by recursively splitting the search area into Z-order sub-regions. The calculation of BIGMIN and LITMAX is used to find where the range should be split.

No further searching is then needed, and the hash (011_2) is added to the result hashes. The same does not apply for the search from 49 to 54, as the quadrants 51 and 52 are outside the search range, but between the Z-order interval. The number of common bits between quadrants 49 and 54 is three (110_2), indicating a vertical split and unknown X-values. The X-bits of quadrant 49 are 100_2, and 111_2 for quadrant 54. They share one common bit, resulting in a BIGMIN mask of 110_2 and a LITMAX mask of 101_2. Interleaving the BIGMIN mask with the Y-bits of quadrant 49 (100_2) yields the BIGMIN value 110100_2 (quadrant 53). Similarly, interleaving the LITMAX mask with the Y-values of quadrant 54 (100_2) yields 110001_2 (quadrant 50). The new search ranges are then from quadrant 49 to 50, and from 53 to 54. The split is indicated by the blue vertical line in Figure 3.4a. Both these new ranges are found to be continuous as they share common bits, followed by all ones and all zeroes. The hashes added to the result are then 11000_2 and 11010_2. However, NASH will only allow continuous ranges which have an even bit length, such that the prefix properties can be utilised in a base 4 encoding. This would result in two further vertical splits, and the hashes for 49, 50, 53, and 54 would be added individually.

Figure 3.4b shows the different levels of the search in a quadtree. Only the visited sub-trees are included. The red lines illustrate the first range split, and the blue lines the second split. The green lines and nodes indicate where the search terminated, and are the hashes added to the result set. The yellow nodes show the quadrants which were not visited as they were within a continuous range. As can be seen, a total of 12 leaf nodes are included in the search, but only 6 are needed to represent the entire range (four if allowing odd hash lengths). If allowing for some precision loss, even fewer hashes could be included in the results. For example, only the hashes for E and C could be included, however, it would come at a cost of decreased search accuracy.

Chapter 4

Implementation Details

This chapter describes the platforms, annotators, and other off-the-shelf tools utilised for implementing NASH.

4.1 Platforms and Annotators

This section describes the tools used to implement NASH.

4.1.1 Source Code

Java [29] is used as the programming language of choice for all implementations of the system. It is a well-established and performant language providing the necessary tools needed to implement NASH. Java SE Development Kit 8 (version 1.8) was used because of its wide compatibility with all needed libraries and frameworks.

4.1.2 Stanford CoreNLP

The Stanford CoreNLP toolkit [30] provides a pipeline for core natural language processing tasks in Java. It is open-source and widely used by both commercial and government actors. The processing pipeline takes raw text as input and runs it through multiple annotators to produce a final set of annotations. These include parts of speech, named entities, dependency parses, co-reference, etc.

Version 4.4.0 of Stanford CoreNLP was used for all needed basic annotations. The Named Entity Recognition (NER) pipeline is the most crucial benefit of this library, providing the recognition of entity types for organizations, numerical values, persons, locations, and temporal values. The temporal interval annotations are made available through the SUTime library [31]. Especially important for the NASH system is the capability of recognising partially specified times, such as "the nineties".

4.1.3 Elasticsearch

Elasticsearch is used to create, store, and query the index. Elasticsearch is a NoSQL-like database built for large amounts of data of different types while being scalable and fault-tolerant [32]. It is built on top of the open-source search library, Apache Lucene [33], providing a set of data structures and methods for text-based search. This means Elasticsearch can provide full-text search from the get-go. Another benefit is the easy-to-use REST (Representational State Transfer) API Application Programming Interface) which makes it straightforward to create complex queries without having to resort to low-level implementations. Elasticsearch also provides a Java client library, making the integration with the rest of the implementation seamless.

4.2 System Overview

The system is built on a client-server architecture as illustrated in Figure 4.1. The client end of the system is a set of modules written in Java, and the server end is an instance of Elasticsearch with custom configurations and mappings. The client is given the task of annotating documents using CoreNLP, calculating Z-order values, and interacting with the database over HTTP (Hypertext Transfer Protocol). This includes the insertion of annotated documents and creating search requests for said documents. The server, or database, is a running instance of Elasticsearch. The server stores all annotated documents and creates the indexes to allow querying of the data. No changes to the Elasticsearch source code are needed, but a custom index mapping was created to define the index fields and characteristics.

4.3 Java Client

The client application has four main tasks represented by individual modules: annotation, indexing, querying, and Z-order curve operations. The annotation and indexing modules are utilised in the pre-processing and indexing of the documents. The query module constructs the query operations, handles query requests to the server, and received the response. The Z-order module is utilised in both the pre-processing and the querying.

4.3.1 Annotation Module

The annotation module is tasked with reading the raw text documents and extracting the needed annotations. CoreNLP is utilised for this purpose. A pipeline is built using the annotators[1]: *tokenize* which tokenizes the text into roughly "words" suitable for further processing, *ssplit* which splits a sequence of tokens into sentences, *pos* which labels tokens with part-of-speech, *lemma* which creates word lemmas

[1]All CoreNLP annotators and descriptions can be found at `https://stanfordnlp.github.io/CoreNLP/annotators.html`

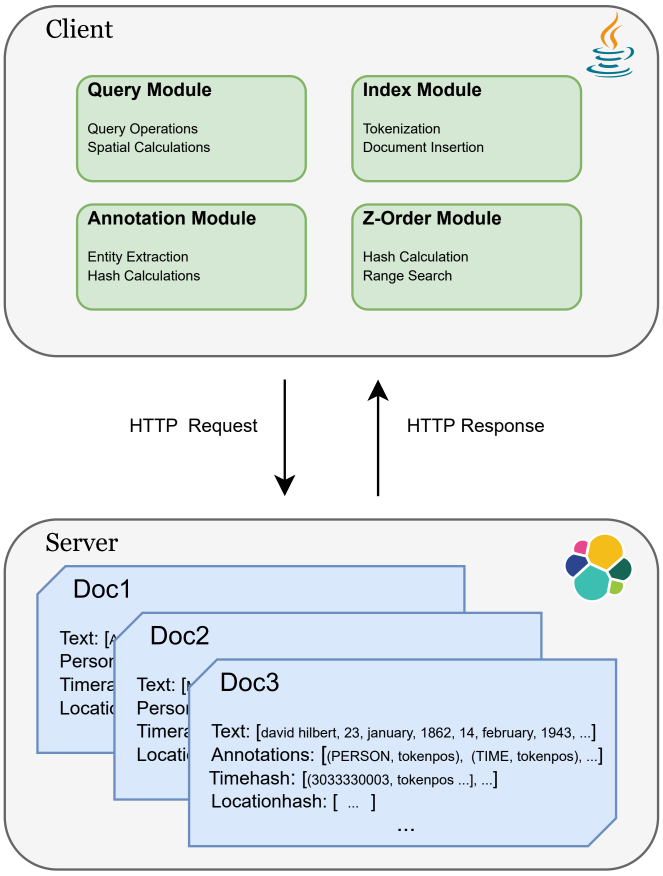


Figure 4.1: The architecture of the full system.

for the tokens, and *ner* which recognizes named, numerical, and temporal entities. The input text moves through one annotator at a time, utilising the output from the annotators before it, ending in the NER annotator.

Entity Extraction

The NER annotator is configured to extract three main types of entities: named (persons, organizations, locations), numerical (money and numbers) and temporal. The SUTime is part of the NER annotator and is configured to mark ranges from the temporal annotations. A text string such as "1990s" will then be annotated as the time interval [1990-01-01, 1999-31-12]. Numerical annotations can include range as well, e.g., "5-7 million dollars" can be marked with the range [5.0E7, 7.0E7]. The entity mentions are also marked with their entity type. The entity types used are those mentioned in Section 3.1. Each annotation also includes the positional span of the tokens the annotation describes.

After extracting annotations using the CoreNLP library, the temporal, numerical, and named location annotations are further processed utilising components from the Z-order module. To be able to support two-dimensional spatial queries over such annotations, they are linearized using Z-order curves over their spatial coordinate system. The entity type annotations do not need further processing.

Numerical Annotations

The numerical annotations can span ranges of large size differences. For instance, one sentence might mention precise measurements at a decimal point precision, while another mentions estimates in the billions. A single Z-order curve covering the entire maximum range of numerical values would have to consist of many bits to be able to get a high enough accuracy (small enough partitions) to distinguish the smaller ranges from each other. Therefore, the maximum range of numerical values is broken into sub-ranges or levels of accuracy. For indexing the documents, four levels are used for the numerical ranges. The first level curve encodes all intervals with the upper bound less than a 100, the second level encodes values more than a 100 and less than 10 000, the third level values more than 10 000 and less than 10 million, and the fourth level all values greater than 10 million and less than a billion. A limit of using the Z-order curve is that there always must be some upper limit, and in this case, every number above a billion is grouped into the same curve. Each level uses a 20-bit precision, but as the maximum range grows larger for each level, the accuracy of the curve gets lower. The size of the most accurate range in each curve can be calculated using Equation 2.3. The first level curve has a maximum accuracy of $\frac{100-0}{2^{10}} = 1 \times 10^{-8}$ in each dimension (beginning and end). The fourth level curve has an accuracy of $\frac{1\times10^{9}-1\times10^{7}}{2^{10}} \approx 1 \times 10^{4}$. However, high valued numbers in text documents are often inherently less accurate than lower value numbers, e.g., "the government spent 66 billion on military defence" has a precision level of $\pm 1 \times 10^{9}$, i.e., it represents a range of [66, 67) billion. This means a high-level accuracy is not needed to sufficiently represent the large numbers. The maximum range for the Z-order curves can also be increased at a higher rate between each level because the accuracy becomes less and less important.

The Z-order hashes are then encoded at base four, meaning two and two bits are represented by a character in the set {0, 1, 2 ,3}. The base 4 encoding makes it possible to fully utilise the hierarchical characteristics by increasing and decreasing the length of the hash while halving the length compared to the bit-representation.

Temporal Annotations

The temporal annotations are also hashed to Z-order values. The bounds of the X and Y dimensions for the temporal values are set to $X, Y \in [1100, 2100]$. Each point in this coordinate system then represents the lower and upper bounds of a numerical interval, just as with numerical annotations. Using a 20-bit precision for the Z-order curve yields an accuracy of $\frac{2100-1100}{2^{10}} = 0.98$, given by Equation 2.3. Thus, each Z-order value represents a two-dimensional range of one-year in each direction. The one-year precision was chosen as a compromise between accuracy and efficiency. The lower and upper bounds for the X and Y dimensions were chosen such that all temporal annotations of recent time will be indexed. If a more precise proximity measure were needed, a similar approach to that of the numerical hashes could be utilised, by using multiple curves with different size ranges. Another approach would be to increase the number of bits in the hash, with the consequence of needing to match longer prefixes. Like the numerical hashes, the temporal hashes are encoded to base 4 strings.

Location Annotations

The location entities returned from the annotation pipeline simply consist of their textual representation, i.e., their names. A separate lookup table of location names and MBRs was used to find the southwest and northeast corner (latitude, longitude) of the smallest rectangular area spanning the location. Country MBRs were gathered from Natural Earth [34]. The cities are not represented by MBRs, but are rather represented by a single point, and were gathered from GeoNames [35] for cities with a population over 500. This is to keep the complexity of the geographical queries at a low level. To find the minimum number of Z-order hashes covering the MBR, a Z-order range search is conducted using the MBR coordinates. The bit-precision for the hashes varies depending on the size of the MBR, with larger areas typically having shorter hashes. The maximum bit-length was set to 20. As the latitude values span an interval from -90 to 90, the accuracy is given by $\frac{90-(-90)}{2^{10}} = 0.18$. The longitude spans an interval from -180 to 180, giving an accuracy of $\frac{180-(-180)}{2^{10}} = 0.35$. Combined, the accuracy is roughly an area of 40 kilometres on Earth.

4.3.2 Z-Order Module

The Z-order Module implements the related algorithms for Z-order curves. This includes hash calculation and range search utilising BIGMIN and LITMAX with optimizations to reduce the number of hashes. The Z-order implementations were made without third-party modules, to have full control over the Z-order values, the precision, and the X and Y dimension boundaries. Most of the operations were implemented through Java bit-wise operators and primitives. The hash itself is represented by the Java primitive long, storing 64 bits in total. The max length of a hash is thus 64 bits using this implementation, however at most 20 bits were used. The algorithms themselves are described in Chapter 3.

Range Search

Searching over the Z-order curve is an important part of the implementation, as it allows for queries over two-dimensional temporal, numerical, and location annotations as a part of the full-text semantic search. The implementation details are covered in Section 3.4. The range search results in a set of hashes which are utilised as part of the queries to the index. The range search is also used when calculating the minimum covering hashes for the MBR of a geographical location used in the indexing module. For location annotations, there are multiple hashes and varying lengths. The maximum precision of the search is defined such that the length of the search hashes is never longer than the length of the hashes which is to be matched.

4.3.3 Index Module

The index module is responsible for preparing the data returned by the annotation module and sending the final data to the server for indexing.

Tokenization

To keep the token positions consistent between the different layers, special considerations must be taken as part of the tokenization process. The documents are tokenized at the client-side to ensure an equal length of the layers and correct token positions. A textual representation of each layer is composed by utilising the text and annotations from the CoreNLP pipeline, which also contains positional information. From these tokens, the different layers are put together. All layers are initialized as lists of the same length, where each position is filled with an *<EMPTY>* token. The tokens are iterated through and placed in the correct layer and position based on their annotation type. For numerical, temporal, and location annotations, the hashes are included in separate hash layers. As tokens can have the same position, multiple layers may have a token at any given position at the same time. For layers which can have multiple tokens at the same position, new tokens are appended and separated by an '&' symbol but are split again when indexed. After each token is iterated through, the resulting layers consist of a set of *<EMPTY>* and text tokens either representing a word or annotation. The layers are then ready to be sent to the Elasticsearch server for indexing.

Algorithm 4 Constructs each of the annotation layers and the text layer.

```
Input: List of text and annotation tokens $T$
Output: The set of all annotations and text layers $L$
 1: procedure CREATELAYERS($T$)
 2:     // Initialize layer token lists with $< EMPTY >$ tokens, length equal to max token
 3:     position of $T$.
 4:     $tl \leftarrow [< EMPTY >, ..., < EMPTY >]$                    ▷ Text layer
 5:     $el \leftarrow [< EMPTY >, ..., < EMPTY >]$                    ▷ Entity type layer
 6:     $th \leftarrow [< EMPTY >, ..., < EMPTY >]$                    ▷ Timehash layer
 7:     $nh \leftarrow [< EMPTY >, ..., < EMPTY >]$                    ▷ Numberhash layer
 8:     $lh \leftarrow [< EMPTY >, ..., < EMPTY >]$                    ▷ Locationhash layer
 9:     for $t$ in $T$ do                                              ▷ Iterate tokens
10:         $p \leftarrow t.position$                                  ▷ integer position of current token
11:         if $t$ is entity type then
12:             $el[p] \leftarrow t.value$
13:         else if $t$ is time then
14:             $th[p] \leftarrow t.value$
15:         else if $t$ is number then
16:             $nh[p] \leftarrow t.value$
17:         else if $t$ is location then
18:             if $lh[p] =< EMPTY >_p$) then
19:                 $lh[p] \leftarrow t.value$
20:             else
21:                 $lh[p] \leftarrow lh[p] + \& + t.value$
22:         else                                                       ▷ Not an annotation
23:             $tl[p] \leftarrow t.value$
24:     $L \leftarrow \{tl, el, th, nh, lh\}$
25:     return $L$
```

Index Insertion

Documents are inserted in the Elasticsearch index after going through the annotation processing and pre-tokenization. The index module reads documents from the disk, creates the text and annotation layers, and sends the layers as part of a JSON document in an HTTP request to the server. The Elasticsearch bulk API is utilised to speed up the insertion. The server can also handle multiple indexing requests in parallel.

4.3.4 Query Module

The query module implements the query operations from Section 3.2 by combining different methods available through the Elasticsearch Java Client API [36], making the Query DSL operations easily accessible through Java. Most importantly is the *Intervals* query. This query allows combining a set of *matching* queries, returning intervals of terms that match the specified queries.

Building the Query Operators

The base building block of the intervals query is the *Match* query. This takes as input a set of terms, the maximum allowed number of positions (max gap) between the tokens, and a Boolean value indicating if the matching tokens need to be in order. The query matches any tokens based on the max gap and the order. This translates to a combination of both the **Boolean Operator** and the **Slop Operator**.

The max gap attribute can also be used to only match within a sentence context, by setting the gap to the same values as the jump of positions between the last token of one sentence, and the first token of the next sentence.

The **Stack Operator** is supported through an *Interval filter* as part of the *Match* query. This allows filtering based on another *Intervals* query and its positional relation to the *Match* query. The interval filter query can be after, before, contained by, contain, etc. in relation to the outer *Match query*. The stack operator is implemented by doing a *Match* query on the phrase which should have an annotation, and filtering with a query on the relevant annotation layer with a “containing” relation. The reverse order is also possible, by having the annotation layer as the outer layer and filtering on the text layer with a “contained by” relation. The *Intervals* query also allows for *Prefix* sub-queries, which are used in combination with hashes to support the **Range Operator**.

Supporting Spatial Queries

The spatial queries of Section 2.3.3, containment, intersect, and proximity, are as explained supported through range searches over the Z-order curve at the client-side. The index lookup at the server-side uses prefix matching, as the indexed values are at maximum precision and often not the same lengths as the search hashes. The prefix queries are combined with the Boolean OR operator for temporal and numerical searches, as only one of the hashes needs to match for it to be retrieved.

4.4 Elasticsearch Server

The server end of the system is an instance of Elasticsearch and provides the indexing, search, and storage capabilities of the system. This section describes how the indexes were configured and implemented in Elasticsearch. An overview of how the query operators from Section 3.2 are defined using queries from the Elasticsearch Query DSL (Domain Specific Language) [37] is also given. Elasticsearch gives the benefit of using already efficient and optimized index structures, as well as a well-developed query language. Additionally, it demonstrates how the indexing of NASH applies to a well-established and popular search engine.

4.4.1 Indexing

Elasticsearch is built on top of Apache Lucene, utilising inverted indexes to support full-text search. As such, there are no needed changes to be made to the underlying database system for it to fit the data and indexing models of the system. The main configurations are done through the mapping, or schema, describing the different fields of the documents, and how they should be analysed when indexed. Data is stored as JSON (JavaScript Object Notation) documents, where a set of keys, or fields, is connected to their corresponding values. Each field describes its type, and how it should be indexed. The different types of fields are indexed using different data structures, which for all fields used in this implementation is the inverted index.

The input JSON documents are produced by the client pre-processing. These consists of the text and the extracted annotations from the CoreNLP pipeline. The final document is produced by the indexing module, consisting of just two main fields. This is the "title" field describing the title of the document, and a "sentences" field. The sentences field consists of an array of text strings, each describing the data layers for that sentence. Since the sentences are separated, a token position jump between sentences can be stored to support searching over sentence regions. As the sentences field is a single text string with all layers, it must be further processed and analysed before indexing. This is defined in the mapping.

4.4.2 Mapping

The full mapping found be found in Appendix A.3. As the query operations require interval queries over multiple fields at a time, the Elasticsearch *fields* attribute must be utilised. The *fields* attribute defines multiple “child ” fields for the “parent” field. Each of the child fields uses the same textual input for indexing as the parent field, but they can utilise different analysers to determine how the input is processed before indexing. The *fields* attribute allows for each of the child fields to be regarded as a single field (but with their own inverted indexes) when it comes to interval queries, such that positional spans between them can be combined.

The parent field is corresponding to the "sentences" field of the input JSON document, and each of the child fields is a different layer in the data model. Each of the child fields has specific analysers which extract only the relevant layer, which is done using regular expressions. The tokens from the index module in the client are separated by a symbol, which in this case is a whitespace character (" ") (it is ensured that no indexed tokens contain whitespace). Therefore, the analyser of each field separates the tokens by splitting the sentence on the whitespace character. For the text layer, no further processing is needed. All the other fields remove the *<EMPTY>* tokens, replacing them with a token without text. The entity type layer is then ready to be indexed as well. The numerical, temporal, and location annotation fields also index prefixes up to the maximum length of a hash. As the location annotations can have multiple entries at the same position, they are split on their separator character.

4.4.3 Searching the Index

The server receives HTTP requests from the client for queries searching over the index. Based on the queries it utilised the inverted indexes for the different fields to match the queries. As all queries are of a Boolean characteristic, scoring of the results is not necessary. The lookup in the index itself and any optimizations happen behind the scenes, meaning the search process mostly functions as a black box. The posting list of matching documents is returned to the client as an HTTP response after the query is complete.

Chapter 5

Evaluation

This section describes the experimental evaluation of NASH. NASH was evaluated to measure the end-to-end query execution time with a set of queries involving temporal and numerical range expressions over three document collections.

5.1 Experimental Setup

This section describes the setup used to conduct the experiments.

5.1.1 Hardware Setup

The experiments were run on a single server-grade machine, equipped with Intel Xeon CPU E5-2640 @ 2.60GHz with 128GB of RAM. Both the client and the Elasticsearch server were running in separate local instances on the machine. Elasticsearch version 8.2 was utilised for the server, with one index shard for each document collection in a single node setup. The Elasticsearch server was given 32GB of heap size. The server was equipped with a set of indexes, implemented as described in Chapter 4 with inverted indexes for text, entity types (person, location, organization, time, number), named entities (persons, locations, organizations), and Z-order hash annotations for two-dimensional range annotations (numerical, temporal, and location). The documents are pre-processed in the client application using Stanford CoreNLP to extract annotations, and Z-order curves to linearize range annotations, before being indexed using the custom mapping defined in Appendix A.3.

5.1.2 Document Collections

Three document collections of varying sizes were used for the experimental tasks. The largest collection is the entire English Wikipedia [38] document collection, which is made public by the Wikimedia Foundation [39]. The second-largest document collection is the New York Times (NYT) annotated corpus [40], consisting of more than 1.8 million news articles over 20 years, from 1987 to 2007. The third document collection is the smallest and consists of all articles from the Simple English Wikipedia [41] also made available by the Wikimedia Foundation [39]. All documents from the three collections are processed through the annotations pipeline at the client-side, before being inserted into an Elasticsearch index for each collection. The benefit of these documents is that they are written using a consistent and well-structured language, and the content is often related to real-world events. The number of documents, the collection and index size are shown in Table 5.1. The table also displays the number of extracted sentences and words from the annotation process. Table 5.2 shows the type and number of annotations in each document collection.

Table 5.1: Document collection sizes, and index sizes in Elasticsearch with number of documents, sentences, and words.

Collection	Size (GB)	Index Size (GB)	Documents	Sentences	Words
EnglishWiki	131.3	140.9	6,472,306	238,175,120	3,228,003,978
NYT	35	32.4	1,878,536	50,424,170	1,088,588,407
SimpleWiki	1.9	1.7	217,116	3,756,866	46,757,033

Table 5.2: The number of annotations of different types resulting from the CoreNLP annotation pipeline.

Collection	Annotations	ORG.	LOCATION	TIME	NUMBER	PERSON
EnglishWiki	685,152,972	87,082,080	99,987,478	177,601,488	115,510,910	204,971,016
NYT	117,274,731	13,585,339	17,095,985	21,001,298	20,567,430	45,024,679
SimpleWiki	7,109,691	916,509	1,289,143	1,840,103	995,099	2,068,837

5.2 Evaluation Task

The system is tested in its capability and efficiency in semantic full-text search with range search over temporal and numerical annotations. The performance is measured in end-to-end query execution time. This includes a set of queries built from the query operations in Section 3.2, combing text phrases and annotations of the different layers. The focus is on the temporal and numerical annotations, to test the performance impact of expanding temporal and numerical ranges used in the search.

5.2.1 Queries

The evaluation query set is gathered from the test bed related to semantic search used by Gupta and Berberich in GYANI [9]. From these, the subset of queries having either a temporal or a numerical expression is gathered. The queries with at least two terms are used. This results in a total of 1698 unique queries, and a sample of 100 queries is used for the benchmark. An excerpt can be seen in Appendix A.5. The queries consist of a set of text phrases and their annotations, combined with the Boolean AND operator. Numerical annotations are represented as single values or intervals, and the temporal annotations are written as intervals within a single year. The annotation types used are PERSON, ORGANIZATION, LOCATION, NUMBER, and TIME. For terms with person and ORGANIZATION annotations, the terms are used as-is and translated to the equivalent Elasticsearch Query DSL operations, using the **Stack Operator**. An example query defined as an Elasticsearch JSON query can be found in Appendix A.4 using numerical and temporal ranges without any text phrases. The queries are translated into the **Span Operator** as defined in Section 3.2, detecting spans in a sentence context. They are executed with a minimal span approach, meaning that only the shortest matching spans are returned. Using the query operators of Section 3.2, an example query can be formulated as $\langle\Delta(TIME)\oplus[easter]\wedge\Delta(NUMBER)\oplus[1,600]\rangle$ This query matches sentences with spans containing the terms "easter" and "1,600", indexed as entity types TIME and NUMBER. To include ranges, the query can be formulated by using the **Range Operator**: $\langle\Delta[1910, 1916, 1919, 1919](TIME)$ $\wedge\Delta[1500, 1500, 1700, 1700](NUMBER)\rangle$, for matching any spans with a temporal expression in the range [1910, 1919], and numerical expression in the range [1500, 1700].

5.2.2 Evaluation Approach

The query set is run in a cold and a warm cache configuration. The warm cache configuration runs a query once without recording the time, before running it three more times in a row for evaluation. The cold cache configuration runs each query once in a random order, which is repeated three times. The warm cache configuration makes so that the relevant documents are kept in main memory (cache), while the cold cache configuration makes it more like that some sections of the index must be loaded from secondary memory. The results show the mean end-to-end query execution time in seconds for the three runs, together with the standard deviation. The same execution approach is used for all query tasks.

5.2.3 Query Tasks

The queries are executed using four different range intervals for numerical and temporal annotations. All query configurations are run with three different Z-order deviation percentages as defined in Equation 3.5, 0.5%, 1%, 10%, and 50%. That is, the ranges are expanded by a percentage in each direction using the *proximity* spatial query. As many of the annotations in the query set have a range size of 0, the range expansions used are a percentage of the maximum ranges for the applicable Z-order curve instead of the input range. For example, for the temporal hashes having a 1000-year maximum range, a 10% expansion equals ±100 years. As the maximum ranges are large compared to most indexed intervals, the percentage extensions are mostly kept at a low level. The 0.5%, 5%, and 10% range extensions represent more realistic queries, while the 50% extension is included to test the performance when the query range almost covers the entire maximum ranges, as at least one of the boundaries would be at the maximum range boundary. Additionally, the intersect query is included to test another more expensive execution. This is run after extending the range by 5% such that it does not only return the contain and containedBy MBRs. The queries run can be summarized as follows:

1. No range extension: percentage deviation not applicable.
2. 0.5% range expansion
3. 5% range expansion
4. 10% range expansion
5. 50% range expansion
6. Intersect with 5% range expansion

Because of limitations in Elasticsearch, only the posting list of matching documents is retrieved, however all (minimal) matching spans are found in the index before the results are returned to the client.

5.3 Results

The results for all the runs are shown in Table 5.3, with the mean execution time over the three runs over the different query tasks. The time is measured in seconds, along with the standard deviation.

5.3.1 Discussion

The results show an increased execution time for larger collections. The Simple Wikipedia collection has a short execution time for all queries without much variation, but it also retrieves by far the least results. This is not very surprising, as the size of the collection is 83 times smaller than English Wikipedia and 19 times smaller than English Wikipedia. This makes it easier for the Simple Wikipedia index to fit entirely in memory. It does however demonstrate that for smaller document collections, the range approach doesn't seem to have a significant performance impact compared to using no ranges. Comparably, English Wikipedia is 4.3 times larger than NYT. The highest execution time for the realistic queries over English Wikipedia is 1.52 seconds for the 10% range query at a 1% deviation. This is also the longest execution time for NYT, at 0.36 seconds. For this query, NYT on average performs the execution 4.2 times faster than NYT, while being 4.3 times smaller. Other queries show similar ratios of execution time and index size, which suggests that the execution times scale linearly.

The intersect query will generally cover the greatest ranges (apart from very large range expansions), as it expands either the lower or upper bound to the maximal range of the dimension. This aligns with the fact that it generally has one of the highest execution times of queries in Table 5.3, even if only tested for a 5%

Table 5.3: Results for the benchmark queries over three different document collections, with warm and cold cache configurations, five levels of range expansion, the intersect query, and two levels of deviation. The mean execution times with the standard deviation are presented in seconds.

Data	Range %	Mean Hits		Cold		Warm	
		1%	10%	1%	10%	1%	10%
English Wikipedia	None	0	0	0.07 ± 0.10	0.07 ± 0.10	0.04 ± 0.05	0.04 ± 0.05
	0.5	352	379	0.18 ± 0.60	0.18 ± 0.63	0.13 ± 0.50	0.12 ± 0.54
	5	782	799	0.30 ± 1.02	0.27 ± 1.04	0.22 ± 0.88	0.21 ± 0.91
	10	940	947	0.34 ± 1.18	0.30 ± 1.12	0.25 ± 1.02	0.23 ± 1.00
	Intersect	908	1000	0.41 ± 1.05	0.35 ± 1.08	0.29 ± 0.89	0.26 ± 0.93
	50	1069	1125	0.39 ± 1.27	0.35 ± 1.28	0.29 ± 1.10	0.26 ± 1.12
New York Times	None	0	0	0.03 ± 0.03	0.03 ± 0.03	0.02 ± 0.01	0.02 ± 0.01
	0.5	37	59	0.04 ± 0.08	0.05 ± 0.10	0.03 ± 0.06	0.03 ± 0.08
	5	183	185	0.08 ± 0.25	0.07 ± 0.26	0.06 ± 0.23	0.05 ± 0.23
	10	206	207	0.09 ± 0.27	0.07 ± 0.26	0.06 ± 0.24	0.06 ± 0.24
	Intersect	234	262	0.13 ± 0.26	0.09 ± 0.26	0.09 ± 0.23	0.07 ± 0.24
	50	217	230	0.10 ± 0.28	0.08 ± 0.28	0.07 ± 0.25	0.06 ± 0.25
Simple Wikipedia	None	0	0	0.01 ± 0.00	0.01 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	0.5	6	6	0.02 ± 0.01	0.02 ± 0.00	0.01 ± 0.00	0.01 ± 0.00
	5	11	11	0.02 ± 0.01	0.02 ± 0.01	0.02 ± 0.01	0.01 ± 0.01
	10	13	13	0.02 ± 0.01	0.02 ± 0.01	0.02 ± 0.01	0.01 ± 0.01
	Intersect	12	13	0.04 ± 0.03	0.02 ± 0.01	0.03 ± 0.02	0.01 ± 0.01
	50	14	15	0.03 ± 0.02	0.02 ± 0.01	0.02 ± 0.01	0.01 ± 0.01

range extension. The number of hits from the intersect query can be affected by the number of indexed interval values in the collection. More intervals compared to points means that the intersect query can potentially make more intersections, outside of those included by the contain and containedBy queries.

The large standard deviation time for most results is caused by the fact that a considerable number of queries will result in no or very few hits, as the Boolean retrieval model does not retrieve partial matches. Looking at the average number of hits, all collections match with an average of 0 (rounded) documents for the no-range query. This implies the sample queries and ranges are too restrictive to find many matches by themselves. Moving only to a ±0.5 percentage increase (for temporal expressions ±5 years) increases the number of documents retrieved significantly.

Furthermore, the results indicate that searching at a high precision level (1% deviation) and a low precision level (10% deviation) generally does not have a meaningful impact on the execution time. This could be caused by the resulting range search hashes not differing much, as the Z-order range search will often terminate before reaching a maximum precision, causing shorter hashes for both approaches. Thus, this could indicate that the other optimizations are working well. The deviation percentage does increase the number of documents received somewhat, with a greater relative increase of hits for the smaller range expansions. This could be caused by the fact that the deviation is determined by the maximum Z-order range and is thus a greater relative increase for the smaller ranges than for the larger ranges. The smaller ranges will also often result in longer Z-order hashes from the range search, as the sub-ranges of the search are less likely to cover entire Z-order regions at lower precision levels (shorter hash). Thus, the lowering of the Z-order precision has a greater impact than for larger ranges where the hashes are more likely to be short, to begin with. It could therefore be beneficial to use a higher accuracy for more precise range searches, however, as the results indicate, the trade-off in retrieval precision might not be worth a minuscule reduction in execution time. Using Z-order curves of a higher precision might lead to more significant results.

If the 1% maximum deviation is interpreted as close to a 100% search recall and precision, the deviation will not impact the recall (Equation 2.2), as the extra hits is a super-set of the more precise results. The search precision (Equation 2.1) is then for the realistic queries between 0.93 and 0.99 for the English Wikipedia collection, and 0.63 to 0.99 for the NYT document collection. The extra hits are however not completely irrelevant in the sense that they are a result of the localisation properties of the Z-order search and will be in close spatial proximity. The range searches can therefore be said to increase the overall recall of the search compared to not using range expansions. The deviation could be used as a rough proximity search, but the results could be skewed by the Z-order curve edge cases.

Limitations of Elasticsearch

The results show the impact of different interval sizes for the range searches, as well as the difference between using a more precise (1%) and less precise (10%). As these are a direct consequence of the Z-order implementation, it makes it difficult to implement sufficient baselines with the same capabilities as NASH, while keeping within the scope of the thesis. Elasticsearch/Lucene brings many benefits in ease of use and optimizations. However, the abstraction level makes it harder to modify and develop query-processing at a deeper level with more control. The current approach is limited by the Elasticsearch Intervals query, which only allows for minimum span matching, and utilising the REST API introduces limitations on transfer sizes of data. Still, this shows some of the usefulness of the Z-order hash approach for range queries. No modifications are needed to the underlying code, and the spatial queries can be implemented in an off-the-shelf search system.

Chapter 6

Conclusion

The goal of this thesis was to develop a system capable of full-text semantic search using spatial queries over numerical and temporal expressions. The indexing system NASH was proposed and implemented, using layers of inverted indexes over annotations and text, based on the work done by Gupta and Berberich [9, 10]. Annotations were extracted from the documents using Stanford CoreNLP [30]. The system indexed annotations for entities of the types: {PERSON, ORGANIZATION, LOCATION, TIME, NUMBER}. The numerical and temporal expressions were indexed as two-dimensional points representing an interval of values. The location annotations were indexed as MBRs. The intervals and MBRs were linearized using Z-order curves, creating a unified data model between all annotation types and text. The spatial queries were implemented using Z-order curves, with custom implementations and optimizations to be able to pre-compute and minimize the Z-order hashes of the range-based search. These were combined with the layered data model, allowing for range search over the inverted index without any modification to the underlying data structures. Using four levels of range extensions for numerical and temporal expressions, and two levels of minimum accuracy, NASH was tested over three data sets of differing sizes. The results show that the proximity-based query scales well over different sized data sets and search range intervals. There is also an indication that the deviation level does not impact the execution performance significantly. The search precision stays high across different search ranges and deviation percentages, which indicates that the optimizations are working as intended.

6.1 Future Work

The work in this thesis allows for spatial queries over temporal, numerical, and location annotations. One area of future work is to extend this type of spatial search to other annotation types. Additionally, the queries using the location annotations could be optimized and evaluated. Named entities such as persons and organizations can be connected to a knowledge base to extract relation information to other entities and categories. For example, a person entity can be of the type "actor", which is an "occupation" etc. This information share some of the same hierarchical properties as the Z-order curve, but the translation to a coordinate system is not as straight forward. This could also allow for other information retrieval tasks outside of semantic querying, such as relation extraction and fact spotting.

Another area for which can be improved upon is to implement a similar indexing approach in a distributed setting. Other types of storage systems, such as column stores could reveal benefits or struggles of the indexing approach. When the data is distributed across multiple nodes, the location preserving properties of the space-filling curve might have a greater impact. Additionally, a lower-level server implementation with more control of the indexing and optimization process could be beneficial to fully utilise the hash indexing. For example, it would be easier to optimize the prefix search depending on the spatial query to be conducted. Some queries only require a single prefix match, while others require all hashes to match, but such restrictions are hard to optimize through high level APIs. With lower-level control, the Z-order range search could perhaps also be included as part of the query processing on the server side, instead of the pre-computation approach used by NASH.

This thesis uses the Z-order curve as the linearizing method to reduce the two-dimensional annotations two one dimension. Optimizations and different approaches of indexing and querying interval values are presented, but other approaches can be implemented and tested as well. For example, the numerical intervals use a tiered Z-order curve approach, and it could be tested with different cut-off points. Additionally, the number of tiers can be decreased/increased, and testing can be done to see what the trade-off between a longer hash-length and more Z-order curves is. The numerical and temporal hashes are indexed using a 20-bit precision, and a longer hash might make the max. deviation optimization more significant. Additionally, other space-filling curves can be tested and compared, such as variations of the Hilbert curve, e.g., compact Hilbert indices [42]. Curves with a better localisation preserving characteristic might reduce the number of splits needed in the range search. Linearization or dimensionality reducing methods other than space-filling curves can also be explored. A comparison of different techniques would be useful for deciding the best approach and see how the approach used in NASH compares.

Bibliography

[1] K. Chang, *Introduction to geographic information systems (4. ed.)* McGraw-Hill, 2008, ISBN: 978-0-07-310171-2.

[2] G. Niemeyer, *Geohash.org is public!*, `https://web.archive.org/web/20080305223755/http://blog.labix.org/#post-85`. (visited on 30/05/2022).

[3] H. Sagan, *Space-filling Curves*. Springer New York, NY, 1994, ISBN: 978-1-4612-0871-6. DOI: `10.1007/978-1-4612-0871-6`.

[4] R. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval - the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011, ISBN: 978-0-321-41691-9.

[5] H. Bast, B. Buchhold and E. Haussmann, 'Semantic search on text and knowledge bases', *Found. Trends Inf. Retr.*, vol. 10, no. 2-3, pp. 119–271, 2016. DOI: `10.1561/1500000032`.

[6] W. Ali, M. Saleem, B. Yao, A. Hogan and A. N. Ngomo, 'A survey of RDF stores & SPARQL engines for querying knowledge graphs', *CoRR*, vol. abs/2102.13027, 2021. arXiv: `2102.13027`.

[7] V. Tablan, K. Bontcheva, I. Roberts and H. Cunningham, 'Mímir: An open-source semantic search framework for interactive information seeking and discovery', *J. Web Semant.*, vol. 30, pp. 52–68, 2015. DOI: `10.1016/j.websem.2014.10.002`.

[8] H. Bast, F. Bäurle, B. Buchhold and E. Haussmann, 'Broccoli: Semantic full-text search at your fingertips', *CoRR*, vol. abs/1207.2615, 2012. arXiv: `1207.2615`.

[9] D. Gupta and K. Berberich, 'GYANI: an indexing infrastructure for knowledge-centric tasks', in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM 2018, Torino, Italy, October 22-26, 2018*, A. Cuzzocrea, J. Allan, N. W. Paton, D. Srivastava, R. Agrawal, A. Z. Broder, M. J. Zaki, K. S. Candan, A. Labrinidis, A. Schuster and H. Wang, Eds., ACM, 2018, pp. 487–496. DOI: `10.1145/3269206.3271745`.

[10] D. Gupta and K. Berberich, ‘Structured search in annotated document collections’, in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining, WSDM 2019, Melbourne, VIC, Australia, February 11-15, 2019*, J. S. Culpepper, A. Moffat, P. N. Bennett and K. Lerman, Eds., ACM, 2019, pp. 794–797. DOI: 10.1145/3289600.3290618.

[11] H. Samet, *Foundations of multidimensional and metric data structures*, ser. Morgan Kaufmann series in data management systems. Academic Press, 2006, ISBN: 978-0-12-369446-1.

[12] R. A. Finkel and J. L. Bentley, ‘Quad trees: A data structure for retrieval on composite keys’, *Acta Informatica*, vol. 4, pp. 1–9, 1974. DOI: 10.1007/BF00288933.

[13] G. R. Hjaltason, H. Samet and Y. J. Sussmann, ‘Speeding up bulk-loading of quadtrees’, in *GIS ’97. Proceedings of the 5th International Workshop on Advances in Geographic Information Systems, November 13-14, 1997, Las Vegas, Nevada, USA*, ACM, 1997, pp. 50–53. DOI: 10.1145/267825.267839.

[14] G. M. Morton, ‘A computer oriented geodetic data base and a new technique in file sequencing’, 1966.

[15] M. F. Mokbel, W. G. Aref and I. Kamel, ‘Analysis of multi-dimensional space-filling curves’, *GeoInformatica*, vol. 7, no. 3, pp. 179–209, 2003. DOI: 10.1023/A:1025196714293.

[16] D. Hilbert, ‘Ueber die stetige abbildung einer line auf ein flächenstück’, *Mathematische Annalen*, vol. 38, no. 3, pp. 459–460, 1891, ISSN: 1432-1807. DOI: 10.1007/BF01199431.

[17] H. Jagadish, ‘Linear clustering of objects with multiple attributes’, *ACM SIGMOD Record*, vol. 19, pp. 332–342, 1990. DOI: 10.1145/93605.98742.

[18] K. Lee, R. K. Ganti, M. Srivatsa and L. Liu, ‘Efficient spatial query processing for big data’, in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, Y. Huang, M. Schneider, M. Gertz, J. Krumm and J. Sankaranarayanan, Eds., ACM, 2014, pp. 469–472. DOI: 10.1145/2666310.2666481.

[19] H. Tropf and H. Herzog, ‘Multimensional range search in dynamically balanced trees’, *Angew. Inform.*, vol. 23, pp. 71–77, 1981.

[20] J. K. Lawder and P. J. H. King, ‘Using space-filling curves for multi-dimensional indexing’, in *Advances in Databases, 17th British National Conferenc on Databases, BNCOD 17, Exeter, UK, July 3-5, 2000, Proceedings*, B. Lings and K. G. Jeffery, Eds., ser. Lecture Notes in Computer Science, vol. 1832, Springer, 2000, pp. 20–35. DOI: 10.1007/3-540-45033-5_3.

[21] M. A. Whitby, R. Fecher and C. Bennight, ‘Geowave: Utilizing distributed key-value stores for multidimensional data’, in *Advances in Spatial and Temporal Databases - 15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21-23, 2017, Proceedings*, M. Gertz, M. Renz, X. Zhou, E. G. Hoel, W. Ku, A. Voisard, C. Zhang, H. Chen, L. Tang, Y. Huang, C. Lu and S. Ravada, Eds., ser. Lecture Notes in Computer Science, vol. 10411, Springer, 2017, pp. 105–122. DOI: `10.1007/978-3-319-64367-0_6`.

[22] S. Nishimura, S. Das, D. Agrawal and A. E. Abbadi, ‘Md-hbase: A scalable multi-dimensional data infrastructure for location aware services’, in *12th IEEE International Conference on Mobile Data Management, MDM 2011, Luleå, Sweden, June 6-9, 2011, Volume 1*, A. B. Zaslavsky, P. K. Chrysanthis, D. L. Lee, D. Chakraborty, V. Kalogeraki, M. F. Mokbel and C. Chow, Eds., IEEE Computer Society, 2011, pp. 7–16. DOI: `10.1109/MDM.2011.41`.

[23] H. J. Haverkort and F. van Walderveen, ‘Locality and bounding-box quality of two-dimensional space-filling curves’, *Comput. Geom.*, vol. 43, no. 2, pp. 131–147, 2010. DOI: `10.1016/j.comgeo.2009.06.002`.

[24] MongoDB, *Geospatial indexes*, `https://www.mongodb.com/docs/manual/core/geospatial-indexes/`.

[25] Elasticsearch, *Geo-point*, `https://www.elastic.co/guide/en/elasticsearch/reference/current/geo-point.html`.

[26] K. Berberich, S. J. Bedathur, O. Alonso and G. Weikum, ‘A language modeling approach for temporal information needs’, in *Advances in Information Retrieval, 32nd European Conference on IR Research, ECIR 2010, Milton Keynes, UK, March 28-31, 2010. Proceedings*, C. Gurrin, Y. He, G. Kazai, U. Kruschwitz, S. Little, T. Roelleke, S. M. Rüger and K. van Rijsbergen, Eds., ser. Lecture Notes in Computer Science, vol. 5993, Springer, 2010, pp. 13–25. DOI: `10.1007/978-3-642-12275-0_5`.

[27] K. Berberich, S. J. Bedathur, T. Neumann and G. Weikum, ‘A time machine for text search’, in *SIGIR 2007: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, July 23-27, 2007*, W. Kraaij, A. P. de Vries, C. L. A. Clarke, N. Fuhr and N. Kando, Eds., ACM, 2007, pp. 519–526. DOI: `10.1145/1277741.1277831`.

[28] A. Anand, S. J. Bedathur, K. Berberich and R. Schenkel, ‘Efficient temporal keyword search over versioned text’, in *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*, J. Huang, N. Koudas, G. J. F. Jones, X. Wu, K. Collins-Thompson and A. An, Eds., ACM, 2010, pp. 699–708. DOI: `10.1145/1871437.1871528`.

[29] Oracle Corporation, *Oracle java*, `https://www.oracle.com/java/`, 2022. (visited on 08/05/2022).

[30] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard and D. McClosky, 'The stanford corenlp natural language processing toolkit', in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[31] A. X. Chang and C. D. Manning, 'Sutime: A library for recognizing and normalizing time expressions.', in *Lrec*, vol. 3735, 2012, p. 3740.

[32] Elastic, *What is elasticsearch?*, `https://www.elastic.co/what-is/elasticsearch`. (visited on 08/05/2022).

[33] Apache Software Foundation, *Welcome to apache lucene*, `https://lucene.apache.org`, 2022. (visited on 08/05/2022).

[34] Natural Earth, `https://www.naturalearthdata.com/`.

[35] GeoNames, `http://www.geonames.org`.

[36] Elastic, *Elasticsearch guide [8.2]*, `https://www.elastic.co/guide/en/elasticsearch/reference/current`. (visited on 08/05/2022).

[37] Elasticsearch Query DSL, `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html`.

[38] Wikipedia: The Free Encyclopedia, `https://www.wikipedia.org`.

[39] Wikipedia Foundation, *Wikimedia downloads*, `https://dumps.wikimedia.org`, 2022.

[40] E. Sandhaus, 'The new york times annotated corpus ldc2008t19', *Linguistic Data Consortium, Philadelphia*, 2008. DOI: `10.35111/77ba-9x74`.

[41] Simple English Wikipedia, `https://simple.wikipedia.org/wiki/Main_Page`.

[42] C. H. Hamilton and A. Rau-Chaplin, 'Compact hilbert indices: Space-filling curves for domains with unequal side lengths', *Inf. Process. Lett.*, vol. 105, no. 5, pp. 155–163, 2008. DOI: `10.1016/j.ipl.2007.08.034`.

Appendix A

Appendix

A.1 Java Implementations

A.1.1 Z-order Hash Calculation

```java
public ZOrderCurve(int bitPrecision, double[] xRange, double[] yRange){
    this.BIT_PRECISION = Math.min(bitPrecision, MAX_BIT_PRECISION);
    this.X_RANGE = xRange;
    this.Y_RANGE = yRange;
    this.X_ERROR = (X_RANGE[1] - X_RANGE[0]) /
    (Math.pow(2, Math.floor(BIT_PRECISION / 2.0) + 1));
    this.Y_ERROR = (Y_RANGE[1] - Y_RANGE[0]) /
    (Math.pow(2, Math.ceil(BIT_PRECISION / 2.0) + 1));
}

public long generateHash(double X_VALUE, double Y_VALUE){
    byte significantBits = 0;
    long bits = 0L;
    boolean evenBit = true;
    //int bitPosition = 0;
    double[] tempXRange = X_RANGE.clone();
    double[] tempYRange = Y_RANGE.clone();
    while (significantBits < this.BIT_PRECISION){
        if (evenBit){
            bits = encodeBit(Y_VALUE, tempYRange, bits);
        } else {
            bits = encodeBit(X_VALUE, tempXRange, bits);
        }
        significantBits++;
        evenBit = !evenBit;
    }
    bits <<= (MAX_BIT_PRECISION - BIT_PRECISION);
    return bits;
}
```

```
protected static long encodeBit(double value,
double[] range, long bits){
    double mid = (range[0] + range[1]) / 2;
    bits <<= 1;
    if (value >= mid){
        bits = bits | 0x1;
        range[0] = mid;
    } else {
        range[1] = mid;
    }
    return bits;
}
```

A.1.2 Range Search

```
public Set<String> rangeSearch(long northWestHash,
                                   long southEastHash,
                                   Set<String> ranges){

    this.calls++;
    int commonBits = commonMSB(northWestHash,
            southEastHash, this.curve.getBIT_PRECISION());


    // The range is in the invalid area
    if (commonBits%2 == 0 && commonBits >1 &&
            checkPrune(northWestHash, commonBits)){
        return ranges;
    }
    // The two corners constitutes a
    // continuous range on the Z-order curve
    if (continuous(northWestHash, southEastHash,
            commonBits) && (commonBits%2 == 0)){
        ZOrderCurve tempCurve = new ZOrderCurve(commonBits,
                curve.getX_RANGE(), curve.getY_RANGE());
        String base4 = tempCurve.toBase4(northWestHash);
        ranges.add(base4);
        return ranges;
    }
    // No more splits possible --> single point on the Z-order curve
    if (commonBits == curve.BIT_PRECISION){
        String base4 = curve.toBase4(northWestHash);
        ranges.add(base4);
        return ranges;
    }

    // If none of the two stop conditions apply
    // we must split the range vertically and horizontally

    long[] nw = bitUntangle(northWestHash, this.curve.BIT_PRECISION);
    long[] se = bitUntangle(southEastHash, this.curve.BIT_PRECISION);

    long litMax;
    long bigMin;
    long[] maxMin;

    // y-value -> horizontal split
    if (commonBits % 2 == 0) {
        maxMin = calculateLitMaxBigMin(new long[]{nw[0], se[0]},
                new long[] {nw[1], se[1]},
                this.curve.BIT_PRECISION/2, 1);
    } else { // X-value --> vertical split
        maxMin = calculateLitMaxBigMin(new long[]{nw[1], se[1]},
                new long[] {nw[0], se[0]},
                this.curve.BIT_PRECISION/2, 0);
    }

    assert maxMin != null;
    litMax = maxMin[0];
    bigMin = maxMin[1];

    rangeSearch(northWestHash, litMax, ranges); // Left/North split
    rangeSearch(bigMin, southEastHash, ranges); // Right/South split

    return ranges;
}
```

A.2 BIGMIN and LITMAX calculation

```
private long[] calculateLitMaxBigMin(long[] knownDimension,
long[] unknownDimension, int bitLength, int dimension){
    long commonMask = 0xFFFFFFFFFFFFFFFFL; // 11111...
    long litMaxMask = 0x7FFFFFFFFFFFFFFFL; //01111...
    long bigMinMask = 0x8000000000000000L; //10000...

    long min = knownDimension[0];
    long max = knownDimension[1];
    long litMaxD;
    long bigMinD;
    int commonBitsCount = commonMSB(unknownDimension[0],
    unknownDimension[1], this.curve.getBIT_PRECISION());
    if (commonBitsCount == bitLength){
        return null;
    }

    // Shifting masks to adhere with the bit_precision and
    // number of common bits
    litMaxMask >>>= curve.MAX_BIT_PRECISION - bitLength +
    commonBitsCount;
    litMaxMask <<= curve.MAX_BIT_PRECISION - bitLength  +
    commonBitsCount;
    bigMinMask = bigMinMask >>> commonBitsCount;

    // Shifting two times in case commonBitsCount = 0,
    // and no shift would happen
    commonMask >>>= curve.MAX_BIT_PRECISION - commonBitsCount -1;
    commonMask >>>= 1;
    commonMask <<= curve.MAX_BIT_PRECISION - commonBitsCount -1;
    commonMask <<= 1;

    litMaxMask >>>= commonBitsCount;

    long commonDBits = unknownDimension[0] & commonMask;

    litMaxD = commonDBits | litMaxMask;
    bigMinD = commonDBits | bigMinMask;

    long litMax;
    long bigMin;
    if (dimension == 0) {
        litMax = bitInterleave(litMaxD, max, bitLength);
        bigMin = bitInterleave(bigMinD, min, bitLength);

    } else {
        litMax = bitInterleave(max, litMaxD, bitLength);
        bigMin = bitInterleave(min, bigMinD, bitLength);
    }
    return new long[] {litMax, bigMin};
}
```

A.3 Elasticsearch Mapping

```
{
  "settings": {
    "analysis": {
      "analyzer": {
        "text_analyzer": {
          "type": "custom",
          "tokenizer": "max_length_whitespace",
          "char_filter":["text_layer"]
        },
        "annotation_analyzer": {
          "type": "custom",
          "tokenizer": "max_length_whitespace",
          "char_filter":["annotation_layer"],
          "filter":["replace_empty_filter"]
        },
        "timehash_analyzer": {
          "type": "custom",
          "tokenizer": "max_length_whitespace",
          "char_filter":["timehash_layer"],
          "filter":["replace_empty_filter"]
        },
        "numberhash_analyzer": {
          "type": "custom",
          "tokenizer": "max_length_whitespace",
          "char_filter":["numberhash_layer"],
            "filter":["replace_empty_filter"]
          },
          "geohash_analyzer": {
            "type": "custom",
            "tokenizer": "max_length_whitespace",
            "char_filter":["geohash_layer"],
            "filter":[
              "replace_empty_filter", "entity_filter",
              "remove_annotations_filter", "remove_null_filter"
            ]
          },
          "entity_analyzer": {
            "type": "custom",
            "tokenizer": "whitespace",
            "char_filter":["entity_layer"],
            "filter":[
              "replace_empty_filter", "entity_filter",
              "remove_annotations_filter", "remove_null_filter"
            ]
          }
        },
        "filter": {
          "timeFilter": {
            "type": "pattern_capture",
            "patterns": ["(TA_[0-3]+)"]
          },
          "entity_filter": {
            "type": "pattern_capture",
            "patterns": ["([^&]+)"]
          },
          "remove_annotations_filter":{
            "type": "pattern_replace",
            "pattern": "(.+&.+)"
          },
          "geohash_filter": {
            "type": "pattern_capture",
            "patterns": ["(LA_[0-9b-hjkmnp-z]+)"]
          },
          "numberFilter": {
            "type": "pattern_capture",
            "patterns": ["(NA[1-4]_[0-3]+)"]
          },
          "annotationFilter":{
            "type": "pattern_capture",
            "patterns": ["(\\((?:PERSON|LOCATION|ORGANIZATION|TIME|NUMBER
                |MONEY)[^)(]*\\))"]
          },
          "replace_empty_filter":{
            "type": "pattern_replace",
            "pattern": "(\\[EMPTY\\])"
          },
          "remove_null_filter": {
            "type": "stop",
            "ignore_case": true,
          "stopwords": [ ""]
        }
      },
      "char_filter": {
        "text_layer":{
          "type": "pattern_replace",
          "pattern": "(1#<(.*)>#1 2#<(.*)>#2 3#<(.*)>#3 4#<(.*)>#4 5
              #<(.*)>#5 6#<(.*)>#6)",
          "replacement": "$2"
        },
        "annotation_layer":{
          "type": "pattern_replace",
          "pattern": "(1#<(.*)>#1 2#<(.*)>#2 3#<(.*)>#3 4#<(.*)>#4 5
              #<(.*)>#5 6#<(.*)>#6)",
          "replacement": "$3"
        },
        "timehash_layer":{
          "type": "pattern_replace",
          "pattern": "(1#<(.*)>#1 2#<(.*)>#2 3#<(.*)>#3 4#<(.*)>#4 5
              #<(.*)>#5 6#<(.*)>#6)",
          "replacement": "$4"
        },
        "numberhash_layer":{
          "type": "pattern_replace",
          "pattern": "(1#<(.*)>#1 2#<(.*)>#2 3#<(.*)>#3 4#<(.*)>#4 5
              #<(.*)>#5 6#<(.*)>#6)",
          "replacement": "$5"
        },
        "geohash_layer":{
          "type": "pattern_replace",
          "pattern": "(1#<(.*)>#1 2#<(.*)>#2 3#<(.*)>#3 4#<(.*)>#4 5
              #<(.*)>#5 6#<(.*)>#6)",
          "replacement": "$6"
        },
        "entity_layer":{
          "type": "pattern_replace",
          "pattern": "(1#<(.*)>#1 2#<(.*)>#2 3#<(.*)>#3 4#<(.*)>#4 5
              #<(.*)>#5 6#<(.*)>#6)",
          "replacement": "$7"
        }
      },
      "tokenizer":{
        "max_length_whitespace":{
          "type": "whitespace",
          "max_token_length": "1048576"
        }
      }
    }
  },
  "mappings": {
    "dynamic": false,
    "properties": {
      "title": {
        "type": "text"
      },
      "sentences": {
        "type": "text",
        "store": false,
        "index": false,
        "fields": {
          "text":{
            "type":"text",
            "analyzer": "text_analyzer",
            "position_increment_gap": 10000,
            "index_phrases": true
          },
          "annotations":{
            "type":"text",
            "position_increment_gap": 10000,
            "analyzer": "annotation_analyzer"
          },
          "timehashes":{
            "type":"text",
            "analyzer": "timehash_analyzer",
            "position_increment_gap": 10000,
            "index_prefixes": {
              "min_chars" : 3,
              "max_chars" : 19
            }
          },
          "numberhashes":{
            "type":"text",
            "analyzer": "numberhash_analyzer",
            "position_increment_gap": 10000,
            "index_prefixes": {
              "min_chars" : 4,
              "max_chars" : 19
            }
          },
          "geohashes":{
            "type":"text",
            "analyzer": "geohash_analyzer",
            "position_increment_gap": 10000,
            "index_prefixes": {
              "min_chars" : 3,
              "max_chars" : 19
            }
          },
          "entities":{
            "type":"text",
            "analyzer": "entity_analyzer",
            "position_increment_gap": 10000
          }
        }
      }
    }
  }
}
```

A.4 Example Query

```
{
  "query": {
    "bool": {
      "filter": [
        {
        "intervals":{
            "sentences":{
              "all_of":{
                "ordered": true,
                "max_gaps": 10000,
                "intervals": [
                  {
                    "match": {
                      "query": "TIME",
                      "analyzer": "whitespace",
                      "use_field": "sentences.annotations",
                      "filter":{
                        "containing":{
                               "prefix":{
                          "prefix": "TA_33",
                          "use_field": "sentences.timehashes"
                          }
                        }
                      }
                    }
                  },
                  {
                   "any_of":{
                     "intervals":[
                        {"prefix":{
                          "prefix": "NA1_03000",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_003",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_01220",
                          "use_field": "sentences.numberhashes"
                        }}
                        ,
                        {"prefix":{
                          "prefix": "NA1_02100",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_02111",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_02110",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_01202",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_01200",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_02110",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_01222",
                          "use_field": "sentences.numberhashes"
                        }},
                        {"prefix":{
                          "prefix": "NA1_02101",
                          "use_field": "sentences.numberhashes"
                        }}
                      ]
                    }
                  }
                ]
              }
            }
          }
        }
      ]
    }
  }
}
```

A.5 Evaluation Queries Excerpt

```
(richard m. nixon)#PERSON ^ (united states)#LOCATION ^
(new york city)#LOCATION ^ (37th)#37.0 ^ (81)#81.0

(pat tillman)#PERSON ^ (afghanistan)#LOCATION ^ (multimillion-dollar)#$1000000.0

(zacarias moussaoui)#PERSON ^ (americans)#MISC ^ (sept.
11)#[2005-09-11 , 2005-09-11]

(bp)#ORGANIZATION ^ (gulf of mexico)#LOCATION ^ (11)#11.0

(george washington)#PERSON ^ (franklin house)#ORGANIZATION ^
(new york city)#LOCATION ^ (first)#1.0

(natchez miss.)#LOCATION ^ (200)#~200.0

(easter)#[1916-04-23 , 1916-04-23] ^ (1,600)#1600.0
```