Large Scale Qualitative Spatio-Temporal Reasoning

Matthew Mantle

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the degree of Doctor of Philosophy

School of Computing and Engineering
University of Huddersfield
September 2021

Abstract

This thesis considers qualitative spatio-temporal reasoning (QSTR), a branch of artificial intelligence that is concerned with qualitative spatial and temporal relations between entities. Despite QSTR being an active area of research for many years, there has been comparatively little work looking at large scale qualitative spatio-temporal reasoning - reasoning using hundreds of thousands or millions of relations. The big data phenomenon of recent years means there is now a requirement for QSTR implementations that will scale effectively and reason using large scale datasets. However, existing reasoners are limited in their scalability, what is needed are new approaches to QSTR.

This thesis considers whether parallel distributed programming techniques can be used to address the challenges of large scale QSTR. Specifically, this thesis presents the first in-depth investigation of adapting QSTR techniques to work in a distributed environment. This has resulted in a large scale qualitative spatial reasoner, ParQR, which has been evaluated by comparing it with existing reasoners and alternative approaches to large scale QSTR. ParQR has been shown to outperform existing solutions, reasoning using far larger datasets than previously possible.

The thesis then considers a specific application of large scale QSTR, querying knowledge graphs. This has two parts to it. First, integrating large scale complex spatial datasets to generate an enhanced knowledge graph that can support qualitative spatial reasoning, and secondly, adapting parallel, distributed QSTR techniques to implement a query answering system for spatial knowledge graphs. The query engine that has been developed is able to provide solutions to a variety of spatial queries. It has been evaluated and shown to provide more comprehensive query results in comparison to using quantitative only techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The basis for this thesis lies in two different areas: qualitative spatio-temporal reasoning (QSTR) and large scale data processing. In this introductory chapter, both topics are described briefly before the motivation for the research is discussed and the fundamental research question of this thesis is presented.

1.1 Qualitative spatio-temporal reasoning (QSTR)

QSTR is a branch of artificial intelligence that is concerned with non-numeric representations of space and time. Rather than working with time as seconds, minutes hours etc. QSTR is concerned with temporal relations between events. For example, consider the following scenario:

> *Bob ate his breakfast while Alice walked the dog. As soon as he finished his breakfast, Bob drove to work.*

This scenario describes a number events e.g. *Alice walked the dog*, and relations between these events e.g. *Bob ate his breakfast* ***while*** *Alice walked the dog*.

Similarly, in the spatial domain, QSTR doesn't rely on pixels, or geometric representations such as points or polygons, instead information is represented as relations between spatial entities. For example, *Huddersfield is* ***in*** *West Yorkshire*, *England* ***contains*** *West Yorkshire* or *West Yorkshire* ***borders*** *Derbyshire*.

Given a collection of qualitative temporal or spatial facts it is possible derive additional information about a scenario, which takes the form of common sense reasoning. Using the spatial scenario described above we can infer that because West Yorkshire is in England, and Huddersfield is in West Yorkshire, Huddersfield is also in England. QSTR can be a little more nuanced than simple transitive relations. For example, if we consider the temporal example from above, we can reason about the relationship between Alice walking the dog and Bob driving to work. Figure 1.1 visually shows the possible relations between these intervals of time. Note that the length of the time intervals is arbitrary, it is only how they are ordered that is significant. Figure 1.1 shows that Alice walking the dog might have *overlapped* the time interval of Bob driving to

work, it might have been *finished by* Bob driving to work (that is they finished at the same time) or Alice walking the dog might have temporally *contained* Bob driving to work. These are the only possible ways these time intervals can be arranged without contradicting the original statements.

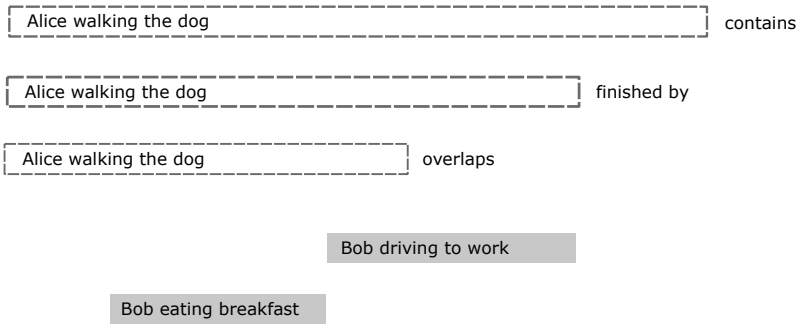


Figure 1.1: Temporal scenario using the Interval Algebra

Qualitative constraint calculi such as Allen's Interval Algebra (IA) and Region Connection Calculus (RCC) provide a formalism for representing these temporal or spatial relations. Specifically they define which relations are possible in a calculus and rules for reasoning using these relations. Chapter 2 describes qualitative constraint calculi in more detail. For example, it provides a complete list of the 13 relations of the Interval Algebra (Table 2.1) and the composition table (Table 2.2) which encapsulates the basic rules for reasoning.

QSTR has been active area of research for many years. The properties of qualitative constraint calculi have been explored in detail, and QSTR has been applied to solve a variety of problems, typically in areas related to planning and scheduling and data analysis.

1.2 Large scale data processing

Since the turn of the century there has been an explosion in the size and complexity of datasets that computing professionals are asked to work with. The origins of these datasets are numerous and varied; for example IoT devices generate vast quantities of data from their sensors, businesses systemically record detailed data concerning their customers' transactions and behaviours, owners of websites log the visits and interactions of their users, social media companies store the details of every post and comment on their platforms.

The scale of these datasets is often so large they can't be processed with traditional database solutions using a single machine. Instead, distributed programming frameworks such as Hadoop [1] or Apache Spark [2] are used to manage the problem of large scale data processing. Distributed programming frameworks split large datasets into separate parts which can be distributed to different machines in a computing cluster and processed in parallel. This allows them to deal with enormous datasets, and process them in timely manner.

[1] https://hadoop.apache.org/
[2] https://spark.apache.org/docs/latest/index.html

1.3 Motivation

Many large scale datasets have a significant spatial and/or temporal element to them. For example, satellites and drones are used by cartographers to generate huge amounts of mapping data, data generated by sensors from IoT devices is time stamped, and photographs, videos, and social media posts are geotagged. Processing spatial and temporal data at scale raises specific challenges including missing and/or imprecise records, integrating heterogeneous information, and dealing with complex spatial/temporal representations. In many ways QSTR is well suited to addressing many of these issues. In order to make the case for processing large scale datasets using QSTR these challenges are explored in more detail below.

Incomplete/inaccurate records Many large scale datasets feature incomplete or inaccurate information. This is partially an issue of scale, the more data, the more chance there is for it to feature errors, but is is also related to how the data is generated and the circumstances they are generated in. For example, IoT devices may suffer from network connectivity issues, sensor errors or power outages that result in missing data [22]. Geospatial data collections maybe incomplete due to cloud cover affecting satellite images or legal restrictions affecting drone access [14]. In natural disaster scenarios such as attempting to monitor flooding or wildfires, a rapidly changing situation often means that accurate complete spatial information simply isn't available [30].

A related problem is imprecise data. For example, the ubiquitous nature of smart phones and IoT devices has made it possible to temporally and spatially locate an individual with high precision. However, in order to protect an individual's privacy, but still allow them to benefit from location based searches, it is often necessary to obfuscate their precise location [18]. Consequently, for privacy reasons, the resulting spatial and temporal records lack precision.

QSTR is especially well suited to working with imprecise and incomplete data. Systems that employ quantitative reasoning i.e. performing calculations to determine temporal and spatial facts, require complete, high precision data e.g. it is difficult to compute if Huddersfield is in England unless we have accurate complete geometric representations of both Huddersfield and England. Conversely, qualitative constraint calculi implicitly support unknown or imprecise information. In the scenario presented at the start of this chapter no information was provided on the relationship between Alice walking the dog and Bob driving to work. However, we could reason using the available, incomplete knowledge to identify possible scenarios and rule out others.

Heterogeneous datasets Data concerning space and time is sometimes only available in qualitative form. This is especially the case if the data originates in natural language form. For example, if asked for the location of Huddersfield, a person is likely provide a qualitative description, 'Huddersfield is east of Manchester' or 'Huddersfield is in West Yorkshire'. Furthermore, in the spatial domain, many vernacular or colloquial places can't be represented as a geometry [67]. For example, Northern England doesn't have precise borders, its location is understood relative to other places e.g. it touches Scotland.

Large scale datasets often feature spatial and/or temporal information in qualitative form e.g. natural language descriptions from social media posts, or linked open data knowledge graphs which feature both quantitative information (object geometries, dates and times) and qualitative spatial information (relationships between entities). These heterogeneous datasets comprising both qualitative and quantitative knowledge cause difficulties for systems reliant on quantitative reasoning, they can only deal with metric information. However quantitative representations can be converted to qualitative relations and combined with qualitative facts to generate complete scenarios which can then be reasoned over using QSTR techniques.

Scale and complexity of datasets Despite the use of distributed programming techniques, the scale and complexity of datasets can still pose challenges for processing. Time-stamps from sensors are recorded at high precision with IoT applications often integrating data from many different sources, resulting in billions of records. High resolution geometric representations can be very large. For example, as we shall see in Chapter 4, regions represented as multi-polygons can be made up of thousands of separate polygons, and millions of coordinates. Even using distributed approaches, processing data of this scale and complexity creates challenges. In comparison, the qualitative representation of spatial data is often lighter weight, making it suited to large scale processing.

1.4 The research question

These arguments suggest that QSTR techniques can have some utility in reasoning over large scale spatial and temporal datasets. However, traditional approaches to QSTR are limited in their ability to scale. Existing state of the art reasoners are only able to reason over datasets featuring hundreds or at most thousands of relations [60]. What is needed are alternative approaches to QSTR that are able to deal with large scale datasets.

An obvious place to look for such an approach is distributed computing. However, despite widespread success in processing large scale datasets, there has been very little published work on the use of distributed approaches to implement QSTR. One possible reason is that distributed computing presents significant challenges. QSTR algorithms need to be fundamentally re-written to work in a distributed environment, and the implementation has to be designed in a way that allows for effective parallelisation so that processing can scale effectively. The above discussion brings us to the central research question for the thesis:

> ***Can parallel distributed computing techniques address the challenges of large scale qualitative spatio-temporal reasoning***

The answer to this question forms the basis of the main contributions which are:

- The development of a novel, parallel distributed QSTR system that is able to reason over large scale datasets and scale effectively

- The formulation of parallel scaleable techniques for generating qualitative relations from large scale complex geometric spatial representations.
- The development of QSTR techniques to provide query answering for large scale heterogeneous spatial knowledge graphs.

1.5 Overview of chapters

- **Chapter 2 - Preliminaries** simply presents the background knowledge needed to understand the main parts of the thesis. This includes a detailed description of qualitative constraint calculi and the fundamental reasoning approach used in QSTR. This is followed by an explanation of parallel, distributed computing frameworks and the key performance challenges faced when processing large scale datasets in a distributed environment. Finally, this chapter ends with a discussion of knowledge graphs.
- **Chapter 3 - ParQR: A large scale qualitative spatio-temporal reasoner** presents ParQR (**Par**allel **Q**ualitative **R**easoner), a novel approach to QSTR that works in a distributed, parallel context. This chapter describes in detail the algorithms used to implement QSTR in a distributed environment. Alternative approaches to the problem of large scale QSTR are considered, and finally ParQR is evaluated using a wide range of datasets, and through a comparison to existing state of the art reasoners.

Chapters 4 and 5 consider a specific application of distributed QSTR techniques - to query large scale knowledge graphs that feature heterogeneous spatial information.

- **Chapter 4 - Enhanced spatial knowledge graph generation** provides the motivation for the development of querying methods for large scale spatial knowledge graphs. It also presents innovative algorithms for generating qualitative spatial relations from quantitative geometric representations. These algorithms are needed to generate a knowledge graph that integrates both quantitative and qualitative spatial information.
- **Chapter 5 - ParQR-QE: A large scale QSTR query engine** presents a distributed spatial query engine, ParQR-QE (ParQR-**Q**uery **E**ngine) that implements QSTR methods to provide solutions to a range of query types. The reasoning approach, which differs in some ways from that presented in Chapter 3, is described in detail along with query execution plans for a range of queries. The system is evaluated using a range of different queries and compared to a quantitative only approach.
- **Chapter 6 - Conclusions** summarises the work from the previous chapters, and reflects on the potential and limitations of parallel distributed programming techniques in the area of large scale qualitative spatio-temporal reasoning

1.6 Publications

The work on ParQR presented in Chapter 3 is largely adapted from the following publications:

- Mantle, M., Batsakis, S., & Antoniou, G. (2016, October). Large scale reasoning using allen's interval algebra. In Mexican International Conference on Artificial Intelligence (pp. 29-41). Springer, Cham. [38]
- Mantle, M., Batsakis, S., & Antoniou, G. (2019). Large scale distributed spatio-temporal reasoning using real-world knowledge graphs. Knowledge-Based Systems, 163, 214-226. [37]

For both publications the contributions are as follows: M. Mantle designed and implemented the algorithms, carried out the experiments, and wrote the manuscripts. S. Batsakis and G. Antoniou guided the work and gave feedback.

Chapter 2

Preliminaries

2.1 Qualitative spatial and temporal constraint calculi

Qualitative spatial and temporal constraint calculi provide a restricted language for describing non-metric information about objects in space and time. The restricted language results in a formalism that can then be used to reason about temporal and spatial information. There are numerous examples of qualitative constraint calculi, the most widely discussed are Allen's Interval Algebra[4], Point Algebra[66] and Region Connection Calculus[49].

2.1.1 Allen's Interval Algebra

Interval Algebra defines 13 possible ways time intervals such as this can be related. These can be seen in Table 2.1, and are referred to as *basic* or *atomic* relations.

Relations in AI are described as being jointly exhaustive and pairwise disjoint (JEPD). That is a pair of time intervals must be related by one of the relations in Table 2.1, and a pair of intervals can be related by only one of the relations. For example, it isn't possible for interval x to be both before and after interval y. Relations in a constraint calculi such as IA each have an inverse. For example, the inverse of *before* is *after*, with the *equals* relation being symmetrical i.e. the inverse of *equals* is *equals*. In a given temporal scenario, the exact relation between intervals may be unknown. In such cases a set can be used to specify possible relations. For example, if x = *Bob brushed his teeth* and y = *Alice read her book* we could state $x\{b, m, d\}y$, interval x takes place *before* interval y, *meets* interval y, or occurs *during* interval y. This is known as a disjunction of relations or a disjunctive relation. If no information is known regarding the relation between a pair of intervals, the relation could be any one of the thirteen basic relations, *{b,bi,m,mi,o,oi,s,si,d,di,f,fi,eq}*. This is known as the universal relation and denoted by the symbol $\mathcal{B}$. Clearly many different relations can be described through different combinations of the 13 basic relations, in total there are $2^{|\mathcal{B}|}$, 2^{13} (8192), possible relations in the IA calculus. The empty relation $\emptyset$ describes a relation that doesn't feature of the basic relations. This has no actual meaning when considering the relation between two intervals but is needed when performing some operations and indicates a consistency issue in a temporal

Relation	Symbol	Inverse	Visual Representation
x *before* y	b	bi	xxxx yyyy
x *meets* y	m	mi	xxxxyyyy
x *overlaps* y	o	oi	xxxx yyyy
x *starts* y	s	si	xxxx yyyyyyyy
x *during* y	d	di	xxxx yyyyyyyy
x *finishes* y	f	fi	xxxx yyyyyyyy
x *equals* y	eq	eq	xxxx yyyy

Table 2.1: The basic relations of IA

scenario. Because the relations are described using sets of atomic relations we can use the usual set theoretic operations, union $\cup$ and intersection $\cap$.

Other qualitative constraint calculi follow the same principles. Two of these are described in below.

2.1.2 Point Algebra

Point Algebra (PA) [66]considers temporal objects as points on a line and is concerned with the three possible relations between time points: precedes ($<$), equals ($=$) and after ($>$)

Point Algebra is a simpler calculi than AI. There are only three basic relations and the total number of relations, $2^{|\mathcal{B}|}$, is eight $(\{<\},\{>\},\{=\},\{<,=,>\},\{<,>\},\{<,=\},\{=,>\},\{\emptyset\})$. The relative simplicity offered by PA results in a number of advantages as we shall see when we consider reasoning problems for qualitative constraint calculi. However, many temporal scenarios, such as the scenario presented in figure 1.1, lend themselves to an interval based, rather than point based representation.

2.1.3 Region Connection Calculus

Whereas as the previous two constraint calculi are concerned with time, Region Connection Calculus (RCC) [49] is concerned with topological relations between regions in space. For example, region x is *connected* to region y.

There are several variants of RCC. The most widely studied is RCC8. The relations for RCC8

are shown in Figure 2.1.

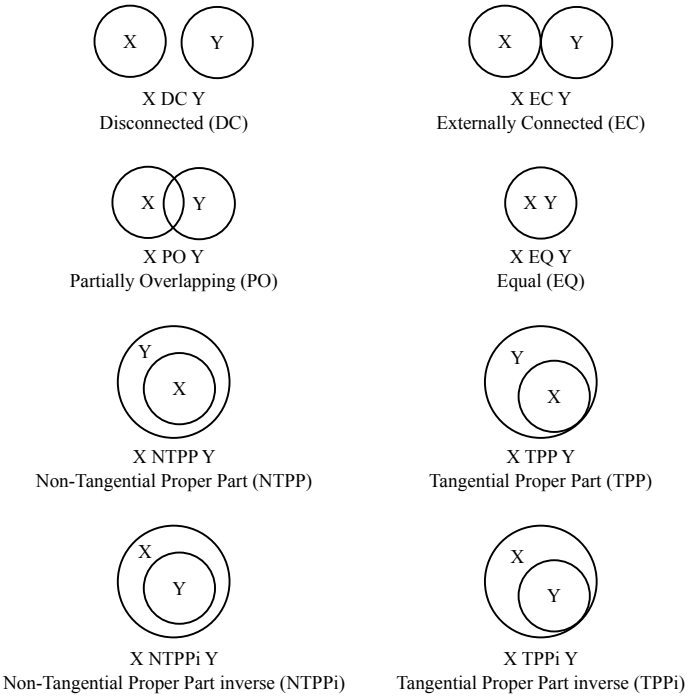


Figure 2.1: The basic relations of RCC8

There are other simpler variants, for example RCC5 where the vocabulary is reduced to 5 relations [15]. Connectedness is ignored, therefore relations such as NTTP and TTP can be reduced to a single Proper Part (PP) relation. Relations in the region connection calculus, like those in Interval Algebra relations have inverses e.g. TPPi is the inverse of TPP. The eight basic relations of RCC8 mean the full calculus features 2^8, 256 relations.

There are other qualitative spatial calculi for example, Cardinal Directions Calculus (CDC) [23] which describes relative positions between spatial entities e.g. equal, north, southeast etc. and the Block Algebra [6] which is a spatial adaptation of Allen's Interval Algebra where the relative position of rectangles whose sides are parallel to the axes of a 2-d Euclidean space can be described using IA relations. These calculi differ from RCC as they are concerned with orientation rather than topology. However they follow the same principles as the calculi described above i.e. they define qualitative relations that can be reasoned over. This thesis focusses on RCC, as does much of the published work on spatial constraint calculi. Reasoning with orientation has challenges e.g. how do we describe the relation between objects where their convex hulls overlap. As a result, orientation calculi lack the easy mapping between natural language expressions and relations that RCC provides [53].

2.1.4 Qualitative constraint networks (QCNs)

A collection of spatial or temporal entities (time intervals, regions etc,) and the relations between pairs of entities form a Qualitative Constraint Network, a QCN. For example, consider the fol-

lowing scenario:

The Yorkshire region has a coastline on the North Sea. The inland city of Leeds is located in Yorkshire. The town of Scarborough is located in Yorkshire. Quebecs hotel is located in Leeds. Quebecs hotel is located in Yorkshire. The Grand Hotel is located in Scarborough.

This scenario describes a number of spatial entities and relations between these entities. The natural language descriptions of the relations can be mapped to RCC8 relations. For example, *Yorkshire {EC} The North Sea* . The more ambiguous spatial adjectives e.g. 'located in' can be mapped to disjunctive relations e.g. *Scarborough {TPP,NTPP} Yorkshire*. A complete list of relations drawn from the above scenario is shown below.

Yorkshire {EC} The North Sea

Leeds {NTPP} Yorkshire

Scarborough {TPP, NTPP} Yorkshire

Quebecs Hotel {TPP,NTPP} Leeds

Quebecs Hotel {TPP,NTPP} Yorkshire

The Grand Hotel {TPP,NTPP} Scarborough

These relations form a QCN, a directed graph where each node represents a variable from the domain, in this case a region in topological space, and directed edges represent the relation between a pair of regions.

z = Yorkshire

y = The North Sea

x = Leeds

w = Scarborough

v = Quebecs Hotel

u = The Grand Hotel

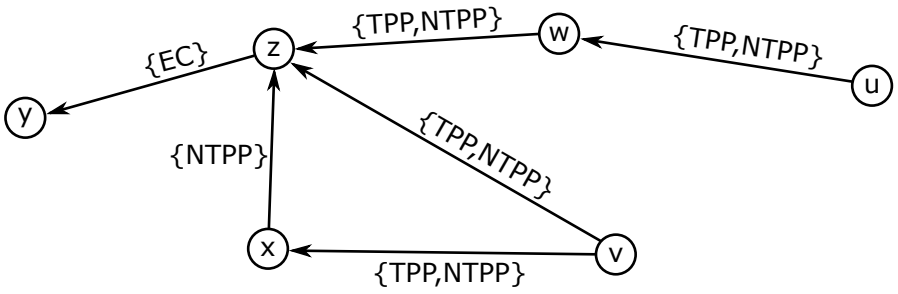


Figure 2.2: Example qualitative constraint network

For simplicity, where a relation is $\mathcal{B}$ i.e. unknown this isn't shown on the network, neither are self relations e.g. (x, x), nor inverse relations.

More formally a QCN can be defined as a directed graph (V, C) where V is a non-empty finite set of variables (time points, intervals, regions etc.). C is a mapping that links each pair of vertices $(v_i, v_j) \in V$ with a relation r, where $r \subseteq \mathcal{B}$. For example, considering Figure 2.2 $C(z, y) = \{EC\}$. Furthermore, for all $v_i \in V$, $C(v_i, v_i) = \{Id\}$, the identity relation i.e the *equals* relation, and for each $v_i, v_j \in V$, $C(v_i, v_j) = C(v_j, v_i)^{-1}$.

2.1.5 Reasoning over qualitative constraint networks

There are a number of reasoning problems associated with QCNs. The most widely studied and important reasoning task is determining whether or not the network is consistent. Is it possible to assign values to the variables without violating any of the relations. In the case of RCC8, can we arrange the regions in such a way that we adhere to all the relations. If we consider the network in Figure 2.2, this is indeed consistent. Figure 2.3 shows some of the possible ways to arrange the regions so that all the relations are obeyed.

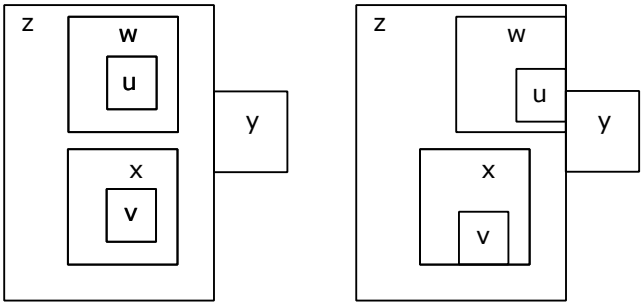


Figure 2.3: Possible consistent instantiations of the QCN from Figure 2.2

Accurately representing the exact shape, position or size of the regions isn't important, we are only interested in the topological relationships between the different regions. Note that if the relationship between *Quebecs Hotel* and *Yorkshire* was *{EC}*, the network would be inconsistent. It wouldn't be possible to arrange the regions so that *Quebecs Hotel* was *{NTPP,TPP}* with respect to *Leeds* and *{EC}* with respect to *Yorkshire*.

Composition

The above example is trivial, and serves simply to explain the notion of consistency. Clearly in more complex networks it wouldn't be possible to visually re-arrange regions/intervals and manually check we have obeyed the relations. Instead we need to consider more robust approaches to reasoning. The key reasoning procedure for QCNs was first introduced by Allen[4] and is based on the idea of composition. If we have relations $C(v_i, v_j)$ and $C(v_j, v_k)$, what are the possible relations that could exist between v_i and v_k. In what way do the relations $C(v_i, v_j)$ and $C(v_j, v_k)$

constrain the relation between v_i and v_k. These compositions are fairly common-sense. For example, if we consider the simple RCC8 network shown in Figure 2.2. the relations $C(x,z)$ and $C(z,y)$ imply a relation between x and y. In fact $C(x,y)$ must be $\{DC\}$. There are no other ways in which we can arrange the regions x and y while still obeying relations $C(x,z)$ and $C(z,y)$. Generally for qualitative spatial and temporal calculi we describe this as *weak composition*, and use the $\diamond$ symbol to denote this. For example, $C(x,y) \diamond C(y,z) \rightarrow C(x,z)$ or as a concrete example from Figure 2.2: $x\{NTPP\}z \diamond z\{EC\}y \rightarrow x\{DC\}y$.

The results of the composition between the basic relations can be described using a composition table, simply a $|B| \cdot |B|$ matrix showing all possible compositions between the basic relations in a qualitative constraint calculi. Part of the composition tables for IA and RCC8 are shown in Table 2.2 and Table 2.3. Full composition tables for the calculi described above can be found in [4] (IA) and [52] (RCC8).

$\diamond$	b	m	o	d
b	{b}	{b}	{b}	{b,m,o,s,d}
m	{b}	{b}	{b}	{o,s,d}
o	{b}	{b}	{b,m,o}	{o,s,d}
d	{b}	{b}	{b,m,o,s,d}	{d}

Table 2.2: Part of the composition table for Interval Algebra

$\diamond$	EC	TPP	NTPP
EC	{DC,EC,PO,TPP,TPPi,EQ}	{EC,PO,TPP,NTPP}	{PO,TPP,NTPP}
TPP	{DC,EC}	{TPP,NTPP}	{NTPP}
NTPP	{DC}	{NTPP}	{NTPP}

Table 2.3: Part of the composition table for RCC8

As noted above, the relation between entities might be a disjunction of basic relations. In which case the composition is simply the union of the composition of all pairs of basic relations.

$$C(v_i,v_j) \diamond C(v_j,v_k) = \bigcup_{a \in C(v_i,v_j), b \in C(v_j,v_k)} a \diamond b$$

Again, referring to the example in Figure 2.2 The composition of the relations between (v,z) and (z,y) can be used to infer the relation between v and y. $v\{TPP,NTPP\}z$ is a disjunctive relation so the composition will be the composition of TPP with EC in union with the composition of NTPP with EC.

$$v\{TPP,NTPP\}z \diamond z\{EC\}y \rightarrow v\{DC,EC\}y$$

Weak composition vs true composition

In the general case the composition operation for qualitative constraint calculi is an approximation and as such is described as being *weak composition*. An example taken from [32] and commonly used to illustrate this idea uses RCC8 and is described below. Consider three regions x, y and z, with $x\{EC\}y$ and $y\{EC\}z$. Using composition and Table 2.3 we can infer the relation between x and z is the disjunctive relation $\{DC, EC, PO, TPP, TPPi, EQ\}$.

$$x\{EC\}y \diamond y\{EC\}z \rightarrow x\{EC, DC, PO, TPP, TPPi, EQ\}z$$

However, if region y has a hole and region x fills this hole (this is still an EC relation) only some of these relations are possible, see Figure 2.4. It isn't possible for the relation between x and z to be EC, PO or TPPi. When the composition is weak, the composition operation only presents us with the strongest implied relation. Of course this is a consequence of region y having a hole and region x filling this hole. When considering simpler regions without holes all the relations implied in Table 2.3 are possible. However, RCC8 imposes no such restrictions on the characteristics of regions, regions with holes and multi-part regions can all be legitimately represented leading to an infinite number of possible relations between entities [50].

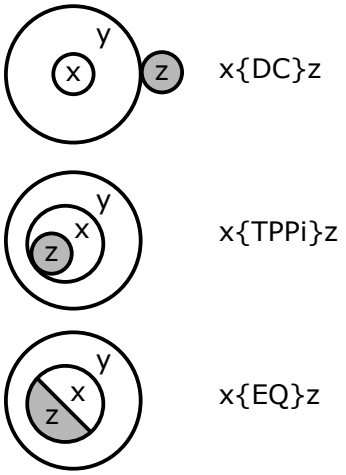


Figure 2.4: Possible relations using weak composition

In simpler more structured domains e.g. time intervals as in the Interval Algebra the composition is in fact true composition (indicated by the $\circ$ symbol) [50]. All relations inferred through composition are achievable i.e. it isn't possible to present time intervals e.g. by changing their duration or position in time in such a way that the initial relations hold and some of implied relations can't be realised.

Algebraic closure

The composition of relations refines the network by narrowing down the possible relations that can exist between variables. Referring to Figure 2.2 there is a relation between v and z, $v\{TPP, NTPP\}z$.

However, the composition of $v\{TPP, NTPP\}x$ and $x\{NTPP\}z$ also implies a relation between v and z, $v\{TPP, NTPP\}x \diamond x\{NTPP\}z \rightarrow v\{NTPP\}z$ (see Table 2.3). By taking the intersection of the existing relation with the newly derived relation we can prune this relation to $\{NTPP\}$, $\{TPP, NTPP\} \cap \{NTPP\} = \{NTPP\}$.

Furthermore the composition of relations adds information into a network that can be used to derive further relations. Again if we consider Figure 2.2, as described above we can use the weak composition operation to infer a relation between u and z, $u\{TPP, NTPP\}w \diamond w\{TPP, NTPP\}z \rightarrow u\{TPP, NTPP\}z$. This new information can form the basis for further reasoning. This new relation can be composed with $z\{EC\}Y$ to infer the relation between u and y, $u\{TPP, NTPP\}z \diamond z\{EC\}y \rightarrow u\{DC, EC\}y$. We can continue in this way, iteratively inferring new relations until a fixed point is reached and no more relations can be derived.

These two operations, iteratively using weak composition to derive relations between different variables, and using intersection to update the relation between variables forms the basis for reasoning over qualitative constraint networks. More formally, iterating using the following operations until a fixed point is reached results in the weak composition closure of the network.

$$\forall v_i, v_j, v_k \in V, C(v_i, v_j) \cap (C(v_i, v_k) \diamond C(v_k, v_j)) \rightarrow C(v_i, v_j)$$

We say that the network is $\diamond$-consistent, *algebraically closed* or *a-closed*. In the event that the intersection of relations derived using different paths is empty($\emptyset$) this indicates that the network is inconsistent. If we reach a fixed point and no more relations can be derived we can conclude that the network is consistent. Figure 2.5 shows this visually for the QCN from Figure 2.2.

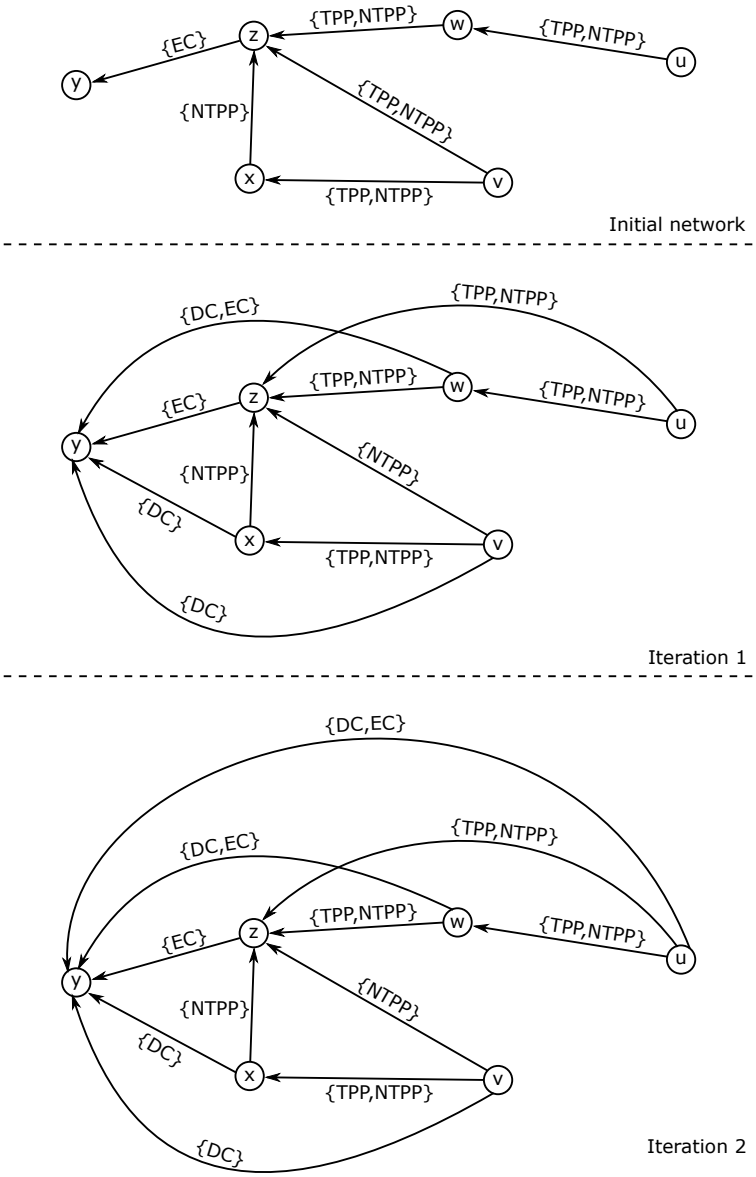


Figure 2.5: Iteratively refining a QCN

For some constraint calculi algebraic closure is equivalent to path consistency, the well known form of local consistency that is used to prune the search space in constraint satisfaction problems. The distinction between $\diamond$-consistency and path consistency is determined by whether or not the composition operation for the calculi is true composition or weak composition. As mentioned previously, if dealing with Interval Algebra the composition is true composition therefore the above formula can be accurately described as path consistency.

Algebraic closure algorithms

Algorithms for deciding consistency in qualitative constraint calculi have their roots in the more general area of constraint satisfaction problems (CSPs). Montanari [39] first introduced the idea of local consistency, checking the consistency of three node triangles in a network as an approxima-

tion of global consistency for CSPs. Mackworth [35] later labelled this path consistency. Checking consistency for QCNs, as described above is simply an adaptation of these techniques to the area of qualitative spatial and temporal networks. However, there is a key difference from the general approaches to determining path consistency in CSPs. Typically CSPs deal with finite domains and the search space is pruned by restricting the possible values variables can hold.

In the case of qualitative constraint calculi we are dealing with infinite domains. For Interval Algebra there are an infinite number of intervals in time. In the case of RCC8 there are an infinite number of possible regions in 2d or 3d space. We are only interested in the relations between entities. Therefore as described above the search space is pruned by filtering the relations between variables, not the values of the variables themselves. Allen [4] first presented a constraint propagation algorithm for Interval Algebra. Vilian et al then described the computational properties of this algorithm [66]. The basic algorithm is applicable to all qualitative constraint calculi and is shown below.

Algorithm 1: Weak consistency algorithm

```
1: Input: N = (V, C), N is a QCN
2: Output: ⋄-consistent N

3: Q ← {(v_i, v_j) | v_i, v_j ∈ V with 0 ≤ i ≤ j ≤ |V|};
4: while (Q is not empty) {
5:     (v_i, v_j) ← Q.pop();
6:     for (k ← 1 to n, k ≠ i and k ≠ j) {
7:         t ← C(v_i, v_k) ∩ (C(v_i, v_j) ⋄ C(v_j, v_k));
8:         if (t = ∅) {
9:             inconsistency detected, end program;
10:        }
11:        if(t ≠ C(v_i, v_k)){
12:            C(v_i, v_k) ← t;
13:            C(v_k, v_i) ← t^-1;
14:            Q ← Q.push((v_i, v_k));
15:        }
16:        t ← C(v_k, v_j) ∩ (C(v_k, v_i) ⋄ C(v_i, v_j));
17:        if (t = ∅) {
18:            inconsistency detected, end program;
19:        }
20:        if(t ≠ C(v_k, v_j)) {
21:            C(v_k, v_j) ← t;
22:            C(v_j, v_k) ← t^-1;
23:            Q ← Q.push((v_j, v_k));
24:        }
25:    }
26: }
```

The input to the algorithm is a QCN where V is simply an array of variables i.e. $V = \{v_1, v_2, ...v_n\}$, and C is mapping between a pair of variables in V and one of the 2^B relations. Consequently C is implemented as a two-dimensional array that stores all relations between all variables in the QCN. The space requirement for such an array is $O(n^2)$, where $n = |V|$. At the start of the procedure Q is initialised to hold the constraints, the pairs of variables. This is typically implemented as a queue structure. In the worst case, where there is a constraint between all pairs of variables in a network, Q has a size of $O(n^2)$. With each iteration of the main *while* loop of

the algorithm, a pair of variables, (a constraint) is removed from this queue. By iterating over all $v_k \in V$, an inner *for* loop performs weak composition for all three variable subsets of the network that feature both variables from this pair.

Lines 7 and 16 perform these operations. As discussed previously there are two parts to this, inferring the possible relation between variables using composition ($\diamond$), and checking consistency by taking the intersection of this newly inferred relation and the existing relation. The resulting relation is stored in a temporary variable. A number of checks then need to be done. If this relation is empty ($\emptyset$), this indicates an inconsistency in the network (lines 8-10 and 17-19), the algorithm has done its job and execution can be halted. If the newly inferred relation isn't empty, a check needs to be done to see if it is stronger than the existing relation (lines 11 and 20). A stronger relation is one that features fewer base relations than the existing relation e.g. if considering Interval Algebra, $\{p, m, o\}$ is stronger than $\{p, m, o, d, fi\}$. If this newly inferred relation is stronger, the QCN needs to be updated with the new relation and the inverse of this relation. Also, because this new relation could form the basis for further inferences, the variable pair for this newly inferred relation is added to the queue. The outer *while* loop runs until there no more pairs of intervals on the queue i.e. no further inferences can be made.

This algorithm runs in $O(n^3)$ time. This is a consequence of the outer *while* loop that runs in $O(n^2)$ and the inner for loop that runs in $O(n)$ time. It is worth considering how these time requirements have been arrived at. The run time of the outer *while* loop is determined by the size of the queue. It is clear that at any point in time there can be at most $O(n^2)$ pairs of intervals (v_i, v_j) on the queue (this is the case if there is a constraint between all variables). However, it should also be clear from Algorithm 1 that pairs of variables can be added to the queue multiple times. Although a pair of variables can be added to the queue multiple times, the number of times a relation can be updated has an constant upper bound. This is dictated by the number base relations in the qualitative constraint algebra, $|\mathcal{B}|$. Every time the relation between a pair of variables is updated, at least one base relation is removed from the set of relations between a pair of intervals. For example, for the Interval Algebra calculus $|\mathcal{B}| = 13$, therefore the maximum number of times a relation can be refined is 13. Seeing as for any qualitative constraint this is always a constant number, the time complexity for this outer loop is $O(n^2)$.

The inner for loop takes a pair of relations from the queue and compares it to other relations in the network, (v_i, v_j), is compared to (v_j, v_k) and (v_k, v_i) for all $v_k \in V$. The size of V is n, therefore this loops runs in $O(n)$. Combining the time requirements of the two loops gives an overall time complexity of $O(n^3)$.

Tractability

An important thing to note about computing algebraic closure for qualitative constraint calculi is that in the general case it is an approximation. The algorithm is sound, it never makes an incorrect inference, all the relations removed from the network cannot form part of a solution. However, it isn't complete. That is the algorithm doesn't guarantee to identify all inconsistencies. Allen

was fully aware of the limitations of his consistency algorithm. In his original paper he provided an example of a QCN that is path consistent, but contained inconsistencies, and for which no solution exists. This idea of the algebraic closure algorithm only providing an approximation is true for many other qualitative constraint calculi e.g. RCC8 [19].

The limitations of the algorithm are a consequence of only ever considering three node sub graphs of the network. In order to develop a better approximation it is possible to consider larger subgraphs. For example, van Beek [9] describes a four-consistency algorithm that considers the relations between sets of four vertices from an Interval Algebra network. Such an algorithm runs in $O(n^4)$ time. In fact it is is possible to get a progressively better approximation by considering larger and larger sub graphs, and more time expensive algorithms. However, for anything other than the smallest networks, this approach quickly becomes unfeasible. In fact, in order to guarantee that all inconsistencies are found in an Interval Algebra network, an exponential time algorithm is required that considers all vertices in a QCN. Vilian et al. [66] provided proof that determining consistency of Interval Algebra networks is indeed NP-complete. The same is true of many other commonly used qualitative constraint calculi including RCC8 [52] and CDC [33].

The implications of this intractability are not as significant as they might first appear. For less expressive algebra such a Point Algebra, computing algebraic closure is sufficient to determine satisfiability for a QCN [66]. Furthermore, for both Interval Algebra and RCC8, there exist tractable subsets of the $2^{|\mathcal{B}|}$ relations for which the $\diamond$-consistency algorithm is able to decide whether or not the network is consistent.

A tractable subset is a fragment of the full calculus that is closed under composition, inverse and intersection. The result of applying any of these operations to a relation in the subset always results in another relation of the same subset. Consequently, as long as the initial relations found in a network all belong to the same tractable set, the $\diamond$-consistency algorithm is sufficient to determine consistency. In order for a tractable subset to be of practical use, it needs to contain all the base relations, as these represent definite knowledge, and the universal relation as this describes relations where we know nothing. Depending on the calculus other relations are often needed e.g. $\{TPP, NTPP\}$ for RCC8, as this represents a more general *contains* relation. There are tractable subsets for both Interval Algebra and RCC8 that meet these requirements. For example, $\widehat{\mathcal{H}}_8$ is a maximal tractable subset of RCC8 containing 148 of the full 256 RCC8 relations [52]. A maximal subset is the largest possible subset that is tractable, adding any further relations to the set would make it intractable. Similarly, there are maximal subsets for Interval Algebra e.g. the $\mathcal{H}_{IA}$ subset [42]. This contains 868 relations, again this includes the 13 base relations and the universal relation.

Minimal qualitative constraint networks

Another key point about consistency checking using the $\diamond$-consistency algorithm is that although it identifies whether or not a solution exists, it doesn't remove all redundant relations from a network. Some relations may remain that can't actually form part of a solution i.e. a disjunctive

relation between two variables that may contain a basic relation which can't participate in a solution. However, if $\forall(v_i, v_j) \in V$ every base relation $b \in C(v_i, v_j)$ can form part of a solution then the QCN is described as being minimal. Clearly a minimal network is a more constrained network, consequently algorithms for computing minimal networks are more computationally expensive than determining $\diamond$-consistency. For example, whereas for the Point Algebra consistency can be determined using the 3-consistency algorithm, computing a minimal network requires a 4-consistency algorithm [9].

Solving qualitative constraint networks

It is also important to note that minimal networks in themselves aren't solutions. They simply identify base relations between variables that can take part in a solution. Some relations can still be disjunctive, it is just that all relations in the disjunction can form part of a solution. In a solution every edge is labelled with a single base relation. In order to identify a specific solution for a QCN, a backtracking algorithm is necessary. By recursively assigning base relations to edges in the network, identifying inconsistencies and backtracking, it is possible to incrementally build up a solution to a QCN. Again there aren't any polynomial time backtracking algorithms that can solve QCNs, it is an NP-complete problem [10].

Although minimal networks and finding solutions (through back tracking) have received attention in the research, most of the focus has been on the consistency problem. This is for several reasons. First, it is often determining consistency that is of most practical use i.e. we often only care if a solution is possible, not what the solutions are. Second, real world networks tend to be made up of basic relations (definite knowledge) and the universal relation (no knowledge). Reasoning about the basic relations often results in another basic relation. For example, there are 169 compositions of basic relations in the Interval Algebra i.e. each of the 13 basic relations in composition with another of the 13. Of these compositions, 97 are basic relations e.g. $\{b\} \diamond \{b\} \rightarrow \{b\}$, the other compositions result in disjunctive relations e.g. $\{m\} \diamond \{d\} \rightarrow \{o, s, d\}$. So reasoning using the $\diamond$-consistency algorithm using the basic relations often results in definite (useful) knowledge being added to the network. Finally, the backtracking algorithms for finding a solution to a QCN rely on consistency checking. $\diamond$-consistency is used as a preprocessing step to restrict the search space, to test the consistency of proposed solutions, and as a forward checking technique [10].

2.1.6 Applications

Broadly speaking, applications of QSTR fall into two areas, planning and scheduling, and data extraction and analysis.

In planning type applications, a plan is created in the form of constraints (relations) between variables in a QCN, reasoning can then be used to check consistency and determine whether or not the plan is feasible. For example, Wallgrün [69] describes the use of the Cardinal Directions

Calculus in mobile robot navigation, specifically map learning. Wallgrün describes a qualitative system with a graph based topological representation of the environment. Based on this map, a QCN is constructed using relations from the CDC e.g. *junction 2 is south west of junction 1*. As the robot moves through the environment and observes places and junctions, constraints are added to the QCN, reasoning takes place, and unfeasible map hypotheses are discarded. This allows the robot to efficiently construct a reliable map of its environment quicker than if qualitative spatial reasoning wasn't used.

An example application of qualitative constraint calculi for data analysis is the use of Interval Algebra in molecular biology by Golumbic and Shamir [24]. They were interested in deciding whether an organism's DNA is linear in structure. Even though the problem domain wasn't directly concerned with time, it was concerned with the linear arrangement of entities, and therefore IA was applicable. Based on experimental data, some pairs of segments within the DNA were found to intersect. However, they only had partial information, and couldn't draw overall conclusions about whether or not the DNA segments were arranged along a line. An Interval Algebra network was constructed where DNA segments were represented as intervals, and experimental results were used to add relations to this network. It was then possible to infer additional relations between segments and draw conclusions about the overall structure of the DNA. For example, a consistent network indicated that the DNA was in fact linear rather than circular in structure.

Further applications of QSTR, specifically in the area of GIS, are considered in Chapters 3, 4 and 5. For additional examples of temporal applications see Bartek [7]. Similarly Wolter and Wallgrün [72] provide a good overview of spatial applications.

2.2 Large scale data processing

As described in the introduction to this thesis, since the turn of the century, it is common for the size of datasets to exceed the capabilities of a single machine. In order to efficiently run computations using these datasets, distributed, parallel approaches to data processing have been adopted. Datasets are split and distributed to different machines in a computing cluster. Each machine in the cluster executes computation in parallel, which allows results to be obtained in a much shorter time.

2.2.1 The basic MapReduce model

There are various models for parallel, distributed computing e.g. MPI [25], OpenMP [16]. The most popular and widely used are based on the MapReduce model [17]. In the basic MapReduce model a program consist of two separate functions. A *map* function and a *reduce* function. The *map* function takes input data and transforms it into key/value pairs. All the key/value pairs with the same key are grouped and this forms the input to the *reduce* function. The *reduce* function takes this list of values and performs a summary operation e.g. counting, finding the average. A simple example is shown in Figure 2.6.

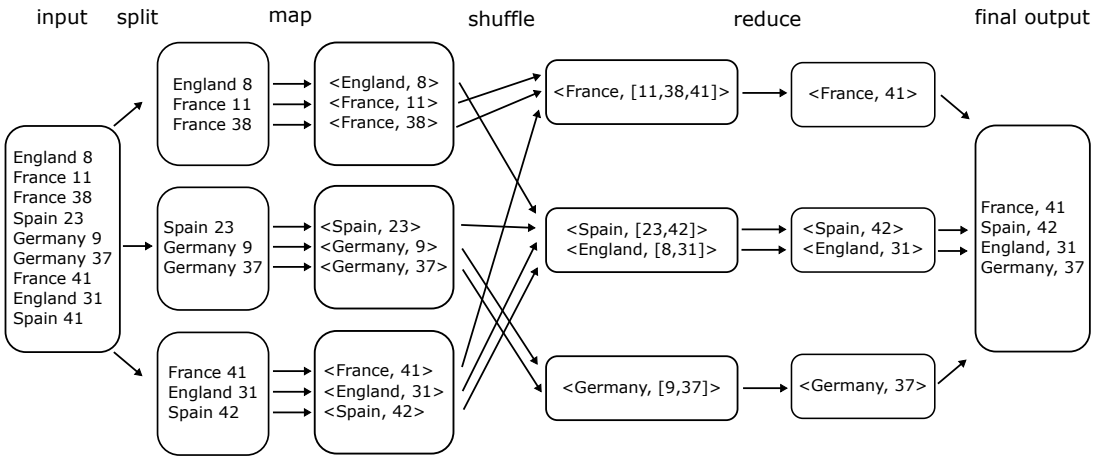


Figure 2.6: MapReduce example

This example computes the highest temperature recorded in a number of different countries. The input to the program is records of different temperatures for different countries. The output is a list of countries and the single highest temperature recorded in that country.

The input to a MapReduce application is typically data stored in the file system e.g. text files. This data is split into a number of input splits or partitions. These are distributed to different machines in the cluster. The *map* operation takes lines of input from it's allocated input split and outputs key/value pairs e.g. $\langle France, 11 \rangle$. All the key/value pairs with the same key are then sent to the same *reduce* function. This involves data being redistributed (shuffled) across the cluster, and then sorted so that all values with the same key are grouped together. The *reduce* function accepts the key and list of values as parameters and outputs a single value. In this case finding the highest temperature associated with the key. These final reduced values are collected to generate the final output. Pseudocode for this simple program is shown in Example 2.1.

Example 2.1: Example MapReduce program

```
map(key, value){
  //key : line number, not needed
  //value : a line from the input e.g. England 8
  parts = value.split(" ")
  country = parts[0]
  temp = Integer.parseInt(parts[1])
  emit(country,temp)
}

reduce(key, values){
  // key : A country e.g.France
  // values: A list of temperatures e.g. [11,38,41]
  maxTemp = values.next().get()
  while (values.hasNext()) {
   temp = values.next().get();
   if(temp>maxTemp){
     maxVal = temp
   }
  }
  emit(key, maxTemp)
}
```

The key point about the programming model is that it allows for problems to be parallelised. Each *map* or *reduce* operation works independently, and the order in which specific map or reduce invocations take place isn't important. Consequently multiple instances of both the map and reduce tasks can be executed simultaneously. By adding more machines to a computing cluster a greater number of *map/reduce* tasks can be run in parallel speeding up the overall execution of a program.

MapReduce implementations

There are many different distributed programming frameworks that implement the model described above. For example, Apache Hadoop [1], Riak [2] and Apache Spark [3]. Importantly, a framework handles many of the tedious aspects of parallel, distributed programming. For example, splitting the input files, distributing data to the different nodes, allocating and scheduling the different tasks, partitioning, grouping and sorting data after the map phase, and collating the final results. Furthermore, frameworks have fault tolerance built in. Worker nodes are monitored, and in the case of a worker node failing, remaining tasks can be re-scheduled on other workers. Specific implementations will also look to optimise execution, for example by scheduling tasks based on data locality, selecting the optimal number of map and reduce tasks and balancing the load between different worker nodes [59]. Although many of these aspects are configurable, or can be overridden by user specified code, essentially the user of a framework writes their own program and the framework handles the lower level details.

2.2.2 The Apache Spark framework

Initially Apache Hadoop was by far most popular implementation of the MapReduce model. However, following its success, a number of more sophisticated frameworks have been developed that provide additional features and benefits for users. These features include richer, higher level APIs, persisting datasets in memory, and relational database type interfaces. These topics are discussed in the following sections where Apache Spark is used to provide concrete examples, and illustrate these ideas. Spark has been chosen as it is the framework used to implement the algorithms presented in Chapters 3, 4 and 5 and because it is considered 'state of the art' in parallel distributed processing. However, it is important to note that many of these features aren't framework specific, similar features are present in other large-scale data processing frameworks.

Richer API

Basic implementations of the MapReduce model such as Hadoop have a limited API consisting of two functions *map* and *reduce*. Although a vast range of different types of applications can be built using these two operations, this can often involve a lot of work for the developer. Spark provides a higher level API with a wider range of operations e.g. *filter*, *join*, *flatMap*, *count*. The

[1] https://hadoop.apache.org/
[2] https://riak.com/
[3] https://spark.apache.org/

following example shows the same 'find the highest temperature' application, but implemented using Spark. In fact the API provided by spark is at such a high level that for this simple example actual code (written in Scala) is shown.

Example 2.2: Simple MapReduce Program Implemented in Spark

```
val input = spark.textFile("data.txt")

//map
val countriesAndTemps = input.map(line=>{
    //line : A line from the input file e.g. England 8
    val parts = line.split(" ")
    val country = parts(0)
    val temp = parts(1).toInt
    (country, temp)
})
countriesAndTemps.cache()

//reduce
val result = countriesAndTemps.reduceByKey((temp1,temp2)=>{
    if(temp1>temp2) temp1 else temp2
})

//output
result.foreach(println)
//France, 41
//England, 31
//Spain, 42
//Germany, 37


//demonstrating some other Spark operations

//counting elements in a dataset
val totalNumOfRecords = countriesAndTemps.count()
println(totalNumOfRecords); //outputs 9


//filtering (where highest temp is greater than 39)
val filteredResults = result.filter(countryTempTuple => {
    countryTempTuple._1 > 39
})

filteredResults.foreach(println)
//France, 41
//Spain, 42
```

The *map* operation remains largely the same, a line from the input is split, and a tuple consisting of the country name and recorded temperature is returned. The *reduce* operation from the MapReduce model is replaced by the equivalent *reduceByKey* operation, where values with the same key are grouped. This *reduceByKey* is a higher-order function that implements a fold on the list of values. Again, just as with MapReduce these operations are parallelised.

The example is expanded on to show some other Spark operations (*foreach*, *count*, and *filter*). The same computations can be achieved using the basic MapReduce model, the richer API simply provides an interface where applications can be written in a more compact style.

Persisting data

Many large-scale data processing algorithms are iterative in nature and need to re-use a dataset across a number of different iterations. For example, machine learning algorithms such as logistic regression require a set of data points to be repeatedly revisited in order to model the probability of an event existing. Similarly, in querying type applications a single dataset needs to be accessed repeatedly in order to provide solutions to different queries. The basic MapReduce model was designed for acyclic data processing, a *map* stage followed by a *reduce* stage. It is possible to implement iterative algorithms in a framework such as Hadoop, however, doing so often involves significant additional running costs, as each cycle runs as a separate job. At the end of a cycle, the data is written to disk, at the start of the next cycle, this data is then read from disk. This incurs costs such as disk I/O and serialisation that can significantly increase execution times [77]. In response to this problem a number of frameworks have emerged that provide greater support for iterative applications e.g. Pregel [36], HaLoop [13]. The Spark framework addresses this issue through the idea of a Resilient Distributed Dataset (RDD) [77]. Essentially, a dataset that can be held in memory and operated on in parallel. Because it can be cached in memory, a dataset can then be easily re-used in an application. Again referring to Example 2.2, the output of the *map* operation i.e. the key/value pairs, is stored in memory (the *cache* operation). This collection of key/value pairs forms the input to the *reduceByKey* operation. There is also a *count* operation that uses the same key/value dataset. However, instead of having to read this data from disk or re-generate the key/value pairs, the dataset can simply be loaded from memory, speeding up execution.

Relational interface

The functional style programming interface offered by the MapReduce model and implementations such as the core Spark API described above are flexible and powerful. However, many data processing frameworks have evolved to provide a more declarative style interface where users can execute SQL queries on distributed datasets e.g. Pig [47] and Hive [65]. The Spark framework also includes a relational interface for distributed data processing in the form of DataFrames [5]. Like an RDD, this is still a distributed dataset, but is structured into named columns, allowing users to interact with it like they would a relational database table. The following example shows the same 'highest temperature' application implemented using Spark DataFrames.

Example 2.3: Relational Interface in Spark

```
//specify the schema for the DataFrame
val fields = List(
    StructField("country", StringType, true),
    StructField("temp", IntegerType, true))
val schema = StructType(fields)


val input = spark.textFile("data.txt")

//Create an RDD from the input
val countriesAndTempsRDD = input
```

```
.map(line=>line.split(" "))
.map(attributes => Row(attributes(0), attributes(1).toInt))

//create the DataFrame using the RDD and schema
val countriesTempsDF = sqlContext.createDataFrame(countriesAndTempsRDD,
    schema)


//using DataFrame operations
countriesTempsDF
    .groupBy("country")
    .max("temp")
    .show()

//+-------+----+
//|country|temp|
//+-------+----+
//|France | 41 |
//|England| 31 |
//|Spain  | 42 |
//|Germany| 37 |
//+-------+----+


//Executing SQL queries
countriesTempsDF.registerTempTable("countries")
sqlContext.sql("SELECT country, max(temp) FROM countries GROUP BY country")
    .show()

//+-------+----+
//|country|temp|
//+-------+----+
//|France | 41 |
//|England| 31 |
//|Spain  | 42 |
//|Germany| 37 |
//+-------+----+
```

Operations can be executed using DataFrame operations as chained procedure calls e.g. *groupBy, max* or as SQL queries. The use of DataFrames offers two advantages for developers. The first is the opportunity to use a more declarative SQL type syntax, which can be more efficient and intuitive. Secondly, Spark offers optimisation of operations on DataFrames. Dataframes are lazy, when subject to a series of operations, these operations aren't executed until specific output actions are called e.g. *count* or *show*. Consequently, there is flexibility in the ordering and implementation of earlier operations such as filtering or joins. This can reduce the volume of data being manipulated and improve the performance of an application. Furthermore, if the data source is columnar, techniques such as predicate pushdown can be used to reduce the volume of data read from input files [5]. There are potential disadvantages associated with the relational abstraction, mainly that the user loses some of the fine grained control that using the procedural core API allows for. However, the two approaches, a higher level relation type interface using DataFrames and the lower level functional style using RDDs are both supported by Spark; they are inter-operable, applications can make use of either or both.

2.2.3 Performance issues

As described above parallel distributed programming frameworks are designed to handle huge datasets. However, depending on the characteristics of the dataset, and the specific operations that need to be implemented, applications can run into performance issues resulting in excessive execution times, and in some cases failure to complete.

Shuffle operations

Often performance issues are related to shuffling, where data is re-distributed across different partitions and different machine in a clusters. If we consider the simple MapReduce example in Figure 2.6, all the values with the same key need to be co-located in order to find the highest temperature for each country. For example, following the map phase, the key/value pairs for France can be found on the first partition and third partition. These need to be merged before the reduce operation can be executed. At this point the shuffle re-arranges the data so that these values will reside in the same partition. This involves disk I/O, data serialisation and network I/O making the shuffle a costly operation.

Joins

One operation that requires data to be shuffled is the join operation. A simple natural join example is shown in Figure 2.7. This depiction simplifies some aspects but is presented to illustrate the problems associated with shuffling and specifically joining datasets in a distributed setting.

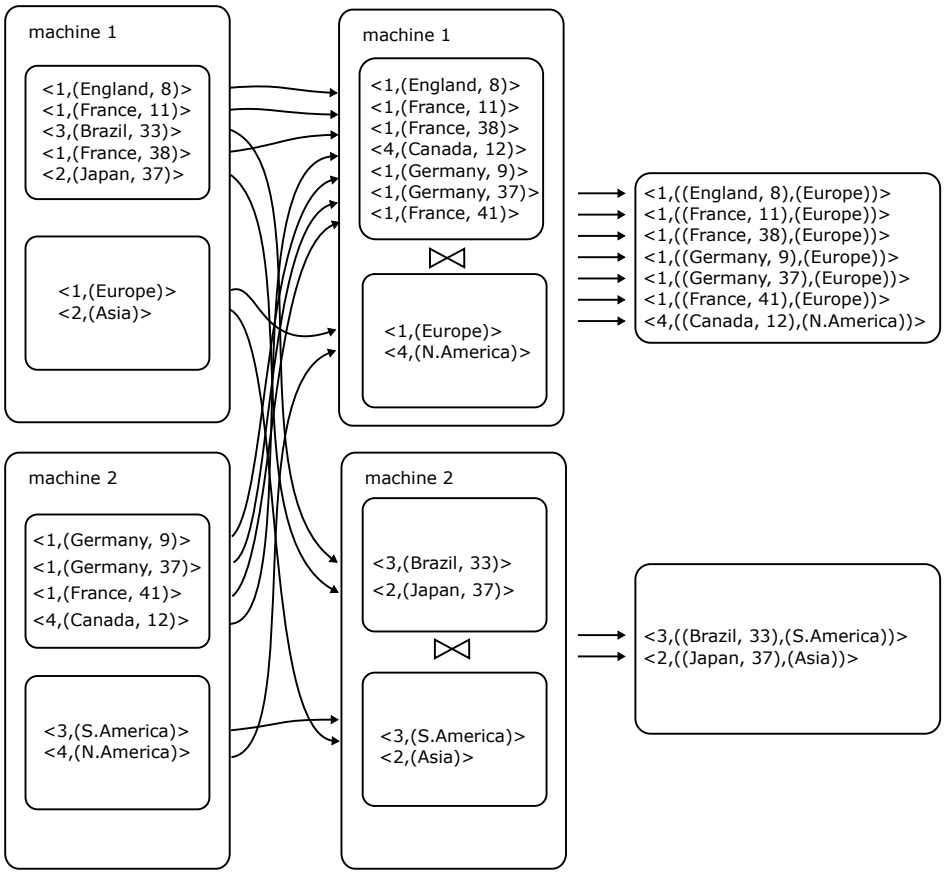


Figure 2.7: Distributed Join

This operation simply joins countries to the continent they are located in. There are two input datasets a country/temperature dataset and continent dataset. Each dataset is in two partitions distributed over two machines in a cluster. The datasets have already been keyed (by a map phase not shown). The key on the countries/temperature dataset is a foreign key that refers to a continent from the continents dataset. In order to execute the join, both datasets need to be shuffled so that all the records with the same key are in the same partition.

Join operations are especially expensive as both datasets need to be shuffled [28]. Furthermore, depending on how the data is distributed, an application can place huge demands on one node in a cluster, leaving others with minimal work to do. In some cases this data skew can largely negate the advantages of a distributed approach. For example, in figure 2.7 there are far more countries from Europe than from any other continent. Consequently, the first machine handles far more work than the second . Clearly this is a trivial example, however, when dealing with much larger, real world datasets, skew can result in a single process being overwhelmed leading to out of memory errors or never ending execution.

There are a number of strategies than can be adopted to improve the performance of joins. These include:

- Where possible, grouping rows to remove duplicate keys prior to the join. For example,

if we also wanted to also find the highest temperature recorded in each country, it would make sense to execute this operation first, thus reducing the size of one of the join inputs.

- Similarly, where possible, early filtering to reduce the size of join inputs e.g. if we only wanted the results to feature countries from Europe.
- Using a dataset that has already been partitioned. If the dataset has already been partitioned using the same key as used in the join, when the join is executed, this data doesn't need to be shuffled again.
- Using a broadcast join strategy, also known as a map-side join or replicated join.

Figure 2.8 shows the same join implemented using a broadcast strategy. In a broadcast join, one side of the join, the smaller of the two datasets, is broadcast to all machines in the computing cluster i.e. a copy of the entire dataset is sent to each machine. In this example the broadcast dataset is the list of continents. Each machine then performs the join using this broadcast variable and the data that is already located at this machine. The advantage is there isn't the need to shuffle the records from the larger side of the join. Furthermore, because all records with the same key no longer need to reside in the same location, the broadcast strategy is more tolerant of data skew. The broadcast strategy does have limitations. It is only really suited to circumstances where the broadcast dataset fits into memory, and is typically used when joining a large dataset with a much smaller one.

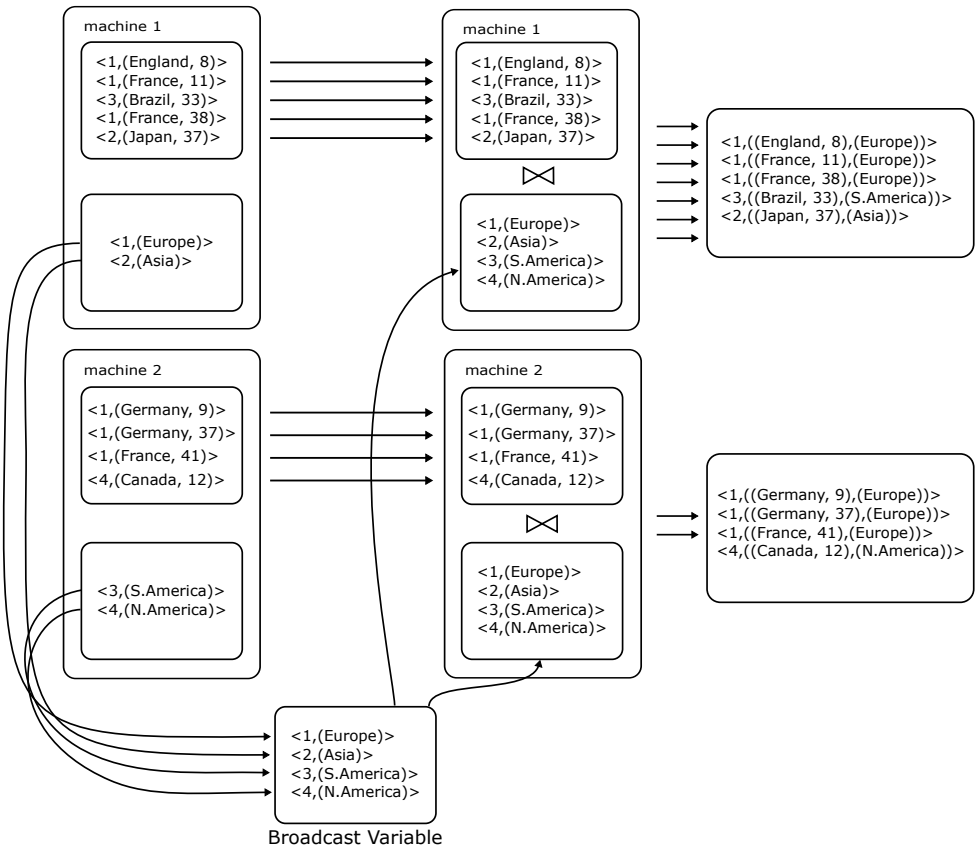


Figure 2.8: Broadcast Join

2.3 Knowledge graphs

There is some argument about the exact definition of a knowledge graph [20], however this thesis uses the term knowledge graph to describe a dataset with the following widely accepted features:

- The dataset has a graph structure where the nodes define entities, and labels on directed edges of the graph represent relationships between entities. Unlike more rigid data representations such as the relational model, the graph structure allows for flexible linking between entities.
- There exist reasoning mechanisms that can be used derive additional facts from the knowledge graph.

Although the term knowledge graph isn't widely used in the QSTR research community, we can view QCNs as a specific type of knowledge graph where the entities are variables from the QCN, labelled edges are relations from the qualitative constraint calculi, and reasoning is implemented using $\diamond$-consistency. Throughout this thesis QCNs are often referred to as knowledge graphs.

2.3.1 Resource Description Framework (RDF)

A commonly used data model for knowledge graphs is RDF. Chapter 4 refers to a number of widely used large scale knowledge graphs, YAGO and DBpedia, which use the RDF data model, so it is worth discussing here. RDF represents a knowledge graph as a collection of statements, also known as triples. Each statement is made up of three parts, a subject, a predicate (or property) and object. Subjects depict entities from the graph, predicates are relations between entities, and objects are either themselves entities i.e. other subjects or literal values. Example 2.4 shows a subset of RDF from the YAGO 4 knowledge graph, presented in the N-Triples format. Each individual line is a statement, with the subject, predicate and object separated by whitespace. The full-stop indicates the end of the statement. Figure 2.9 shows the same data visually as a graph.

Example 2.4: RDF Statements in N-Triples Format

```
<http://yago-knowledge.org/resource/Belgium> <http://schema.org/
    foundingDate>  "1830-10-04"^^<http://www.w3.org/2001/XMLSchema#date> .
<http://yago-knowledge.org/resource/Belgium> <http://schema.org/geo> <geo
    :50.64,4.67> .
<http://yago-knowledge.org/resource/Belgium> <http://schema.org/memberOf> <
    http://yago-knowledge.org/resource/European_Union> .
<http://yago-knowledge.org/resource/Belgium> <http://www.w3.org/1999/02/22-
    rdf-syntax-ns#type> <http://schema.org/Country> .
<http://yago-knowledge.org/resource/European_Union> <http://www.w3.org
    /1999/02/22-rdf-syntax-ns#type> <http://yago-knowledge.org/resource/
    Political_organisation> .
<http://yago-knowledge.org/resource/France> <http://www.w3.org/1999/02/22-
    rdf-syntax-ns#type> <http://schema.org/Country> .
<http://yago-knowledge.org/resource/Belgium> <http://schema.org/memberOf> <
    http://yago-knowledge.org/resource/European_Union> .
```

In the first statement of Example 2.4, the object is a literal value - *"1830-10-04"*. Alternatively, the object of a statement might itself be a subject. For example, the object of the third RDF statement

```
<http://yago-knowledge.org/resource/European_Union>
```

is the subject of the fifth statement. As we can see in figure 2.9, linking entities together in this way, gives RDF it's graph structure.

RDF uses globally unique ids for subjects, predicates and datatypes e.g.

```
<http://yago-knowledge.org/resource/Belgium>
```

These unique ids are known as Uniform Resource Identifiers (URIs). By defining identifiers globally, knowledge graphs can reference and link to entities in separate knowledge graphs without the complication of name conflicts.

Namespaces can be used to group URIs. For example, *foundingDate* and *containedInPlace* are both part of the *http://schema.org* namespace. The N-Triples format uses fully qualified URIs i.e the namespace followed by the subject/property/object. However, it is often more convenient and clearer to use a prefix when referring to entities e.g. *yago:Belgium* as used in Figure 2.9.

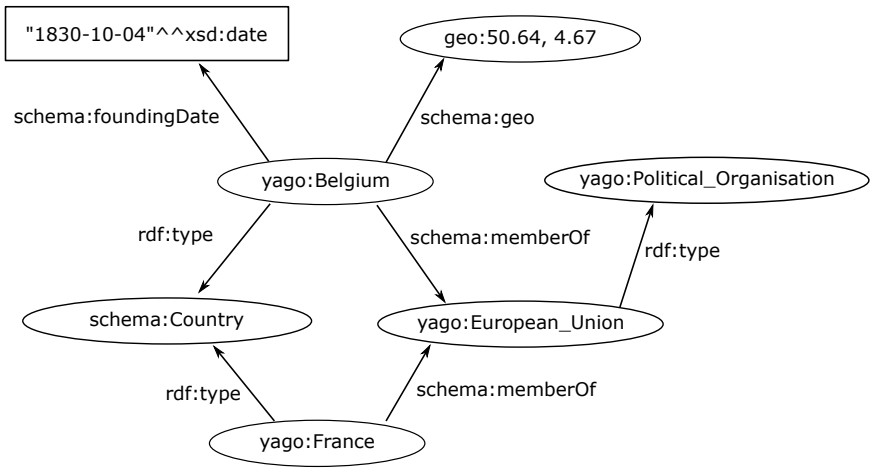


Figure 2.9: Example Knowledge Graph

Reasoning using RDF isn't considered in any detail in this thesis, the focus is solely on QSTR. However, for completeness it is worth explaining that reasoning using RDF data is typically accomplished using the Web Ontology Language (OWL) or RDF Schema, which allows authors to create ontologies in RDF which can be used to formally define classes, their properties and hierarchical structure. These ontologies can be used to reason about RDF data and derive additional facts.

2.3.2 Querying RDF data

RDF knowledge graphs can be queried using an SQL type query language known as SPARQL, an example is shown in Query 2.1 which finds countries that were founded on the 4th of October 1830. Using the knowledge graph from Figure 2.9 this query would return a single result, *yago:Belgium*.

Query 2.1: Example SPARQL Query

```
PREFIX http: <http://www.w3.org/2011/http#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX sc: <http://purl.org/science/owl/sciencecommons/>
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?country
   WHERE {
      ?country rdf:type schema:Country.
      ?country schema:foundingDate  "1830-10-04"^^xsd:date
   }
```

A SPARQL query is made up of a number of triple patterns e.g.

```
?country rdf:type schema:Country.
```

These triple patterns are like RDF statements, but a variable can be used in place of the subject, predicate or object. Variables are denoted with a question mark prefix e.g. *?country*. The set of all triple patterns in a query is referred to as a basic graph pattern. Queries are executed by matching this basic graph pattern with the knowledge graph being queried and substituting the variables for subjects/objects/predicates from the graph [68]. This is a simple example, in addition to basic graph pattern matching, SPARQL supports other SQL type operations e.g. aggregation, ordering and filtering.

2.3.3 GeoSPARQL

GeoSPARQL is a standard built on top of SPARQL and designed to support the spatial querying of RDF knowledge graphs [46]. The standard has been developed by the Open Geospatial Consortium (OGC) and it has specifies two things:

1. A specification for representing geospatial data in RDF. This includes a simple ontology for describing spatial entities and requirements for how geometric data should be encoded in RDF.

2. A query interface for executing spatial queries.

In order to allow more flexibility in terms of how spatial data is encoded in a knowledge graph, the work done in this thesis doesn't comply with the GeoSPARQL standard in terms of representing spatial data in RDF. However, in Chapters 4 and 5 spatial querying of knowledge graphs is considered in some detail. To test the query engine systems that have been developed, and show wider applicability, valid GeoSPARQL queries are used.

Query 2.2: Example GeoSPARQL Query

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?country
   WHERE{
     ?country rdf:type schema:Country.
     FILTER(geof:sfWithin(?country,"POLYGON((4.321423 50.5882119,4.6269803
    50.5869041,4.625607 50.7435868,4.3207364 50.7448903,4.321423
   50.5882119))"^^geo:wktLiteral))
  }
```

Query 2.2 shows an example GeoSPARQL query that will find all the countries that are located within a specified query polygon. In order to execute this query using the example knowledge

graph in Figure 2.9, a query engine could use the *schema:geo* property to determine whether or not *yago:Belgium* is a solution. GeoSPARQL supports a wide range of spatial filters and predicates. For example, as we shall see in Chapter 5, it is possible to query on the basis of spatial relationships between entities such as adjacency and containment.

Chapter 3

ParQR: A large scale qualitative spatio-temporal reasoner

This chapter describes an implementation of the $\diamond$-consistency algorithm for use in a parallel distributed environment. This application has been named ParQR (**Par**allel **Q**ualitative **R**easoner), the full details concerning its working are described below.

3.1 Overview of QSTR using ParQR

Figure 3.1 shows a simple RCC8 network. This is a subgraph of the example shown in 2.1.4. The fundamental approach to reasoning using ParQR is shown in Figure 3.2 and uses this RCC8 network as the example.

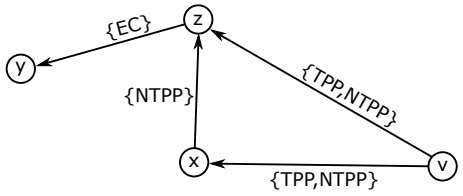


Figure 3.1: Simple QCN

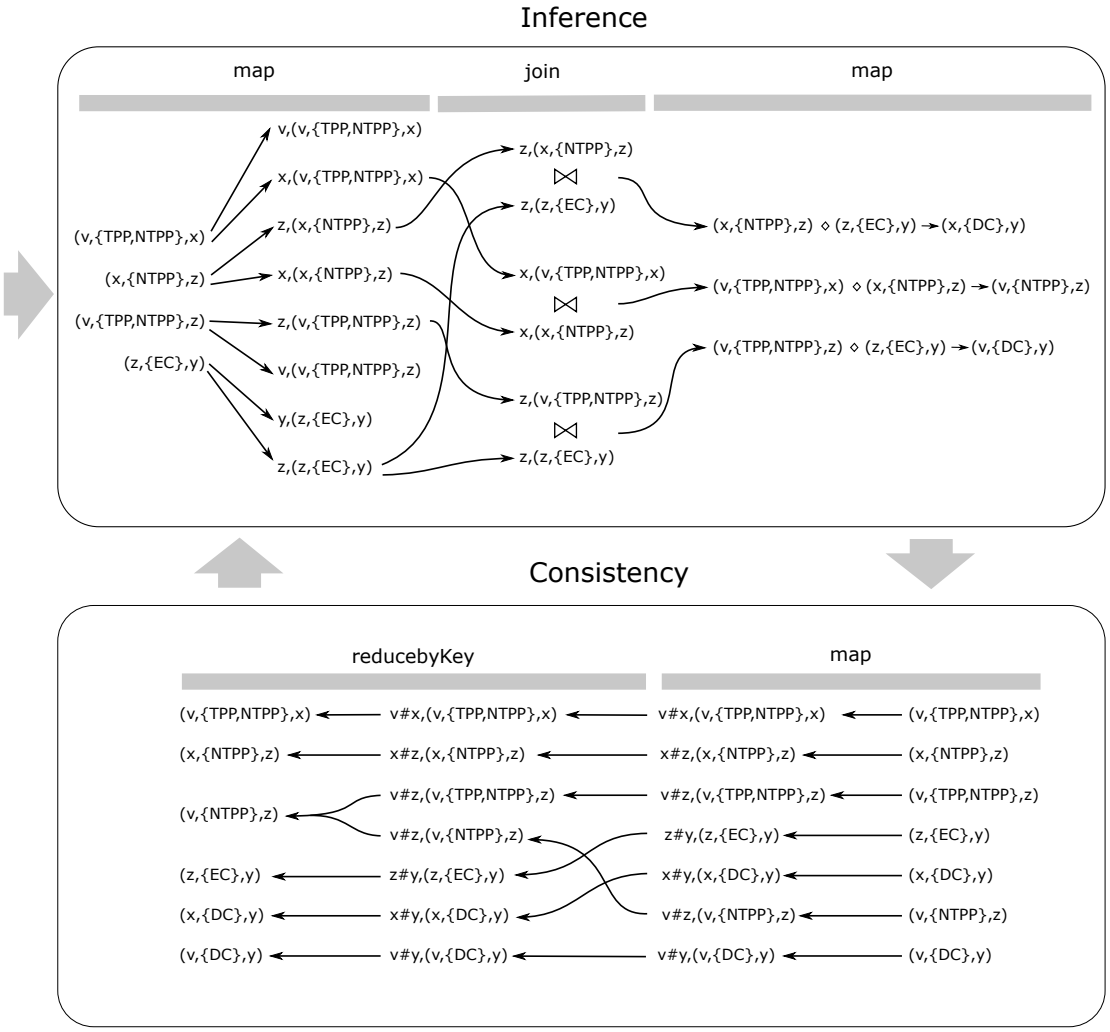


Figure 3.2: Overview of reasoning using ParQR

Computing algebraic closure using ParQR requires two stages, inference and consistency. Inference involves deriving relations between variables in a QCN using the weak composition operation, and consistency is used to compare and update relations following inference.

Edges from a QCN are represented as tuples in the form $(headnode, relation, tailnode)$ e.g. $(z, \{EC\}, y)$. Considering inference first, this involves joining edges that share a variable. In order to execute this join, a *map* operation is used to output keyed tuples. This is done twice. First with the head node as the key, and a second time with the tail node as the key. Then, tuples with matching keys are joined. For example, in Figure 3.2 the edge $(x, \{NTPP\}, z)$ is joined with the edge $(z, \{EC\}, y)$. Using these joined edges we can then use a composition table e.g. Table 2.3 to look-up the inferred relation e.g. $x\{NTPP\}z \diamond z\{EC\}y \rightarrow x\{DC\}y$.

Once these relations have been inferred it is necessary to check for consistency. That is we take the newly inferred relations and compare them to the existing relations to identify inconsistencies. This is implemented by first using a *map* operation to output a key for each edge in the network. For example, all the edges that have v as the head node and z as the tail node will have a key

of *v#z*. Then a *reduceBykey* operation is used to group all the edges with same key and output a single edge for the group. In the simple example shown in Figure 3.2 only one of the groups has more than a single edge, the *v#z* group. In more complex QCNs, there would be many large groups that need reducing. The *reduce* operation involves finding the intersection of the relations in the group. For example, for the *v#z* group in figure 3.2, $\{TPP, NTPP\} \cap \{NTPP\} \rightarrow \{NTPP\}$, which results in the edge $(v, \{NTPP\}, z)$. If the intersection results in an empty set this indicates an inconsistency in the network. If there are no inconsistencies, the output from the consistency stage feeds back into the inference stage, where the new relations can form the basis for further inferences. The two stages, inference and consistency, continue iteratively until a fixed point reached.

Figure 3.2 shows how these operations can work effectively in a distributed setting. Once the input dataset has been split and distributed to different machines in a computing cluster, operations such as using a *map* to generate a key for a tuple, look-up compositions, or find the intersection of a group of relations can be executed in parallel, allowing reasoning to execute quickly using large datasets.

3.2 Limiting the size of joins

Figure 3.2 is also useful for showing the performance bottlenecks of the reasoning process. As described in Section 2.2.3 shuffle operations, that require data to be duplicated and copied across different partitions and machines in the cluster, are costly operations. In Figure 3.2 we can see that a shuffle is required at two points. During inference, edges with same key need to reside in the same partition so they can be joined, therefore a shuffle is invoked to move these edges. Similarly, in the consistency stage, all edges between the same pair of variables need to be moved to the same partition for the *reduce* operation to execute.

Even using high memory computers (e.g. >30GB RAM), for large scale QCNs consisting of millions of relations, a naive implementation such as that shown in Figure 3.2 would quickly occupy the available memory. The join at the inference stage would grow larger with each subsequent iteration. For example, consider the IA network shown in figure 3.4 that consists of a single chain.

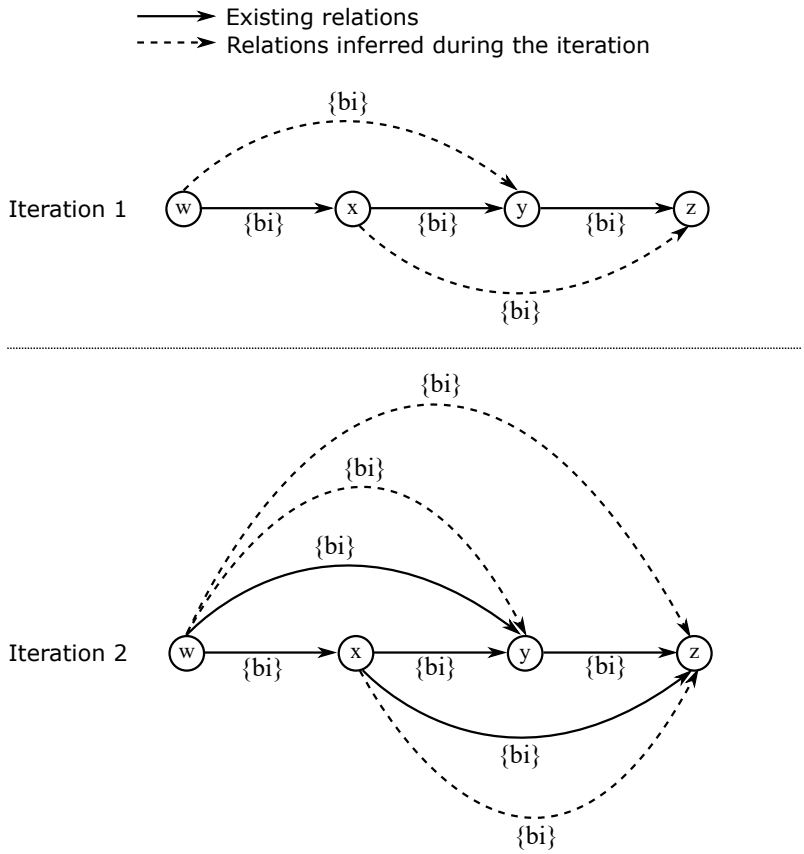


Figure 3.3: Duplicate inferences between iterations

At iteration 1, relations between nodes w and y and between the nodes x and z are inferred. At iteration 2 an inference between nodes w and z is made. However, at iteration 2 we would also join $(w, \{bi\}, x)$ to $(x, \{bi\}, y)$ and re-compute the same bi relation between w and y that was made at iteration 1. Clearly, as we follow longer and longer chains of inference we will generate an increasingly large join in the inference stage that soon becomes unmanageable.

Similar problems have been studied previously. For example, Afrati et al [2] consider recursive algorithms such as computing transitive closure in distributed environments. They identify algorithms that guarantee that paths are only followed once. Similar approaches have been adopted in ParQR to limit the duplicate inferences between different rounds.

In order to limit duplicate derivations, it is necessary to track the distance between nodes in the QCN. This distance between the nodes of an edge is then used to determine which edges will participate in the join at a given iteration. There are various ways to do this. One strategy is to use a linear join strategy where one side of the join has edges with a distance of one. The other side of the join features edges where the distance between nodes is equal to the iteration number. At iteration 1, the QCN is joined with itself. The edges output from this iteration will have a distance of 2. At iteration 2, one side of the join will feature these newly inferred edges, with the input QCN (edges with a distance of 1) as the other side. By limiting which edges can be joined, duplicate derivations are avoided, see Example 3.5.

Example 3.5: Using a linear strategy

```
Input QCN
(u,{bi},v,1)
(v,{bi},w,1)
(w,{bi},x,1)
(x,{bi},y,1)
(y,{bi},z,1)

Iteration 1
(u,{bi},v,1) ◇  (v,{bi},w,1) → (u,{bi},w,2)
(v,{bi},w,1) ◇  (w,{bi},x,1) → (v,{bi},x,2)
(w,{bi},x,1) ◇  (x,{bi},y,1) → (w,{bi},y,2)
(x,{bi},y,1) ◇  (y,{bi},z,1) → (x,{bi},z,2)

Iteration 2
(u,{bi},w,2) ◇ (w,{bi},x,1) → (u,{bi},x,3)
(v,{bi},x,2) ◇ (x,{bi},y,1) → (v,{bi},y,3)
(w,{bi},y,2) ◇ (y,{bi},z,1) → (w,{bi},z,3)

Iteration 3
(u,{bi},x,3) ◇ (x,{bi},y,1) → (u,{bi},y,4)
(v,{bi},y,3) ◇ (y,{bi},z,1) → (v,{bi},z,4)

Iteration 4
(u,{bi},y,4) ◇ (y,{bi},z,1) → (u,{bi},z,5)
```

There are other strategies that can be used to prevent the same derivations being made repeatedly. The smart strategy [2] executes a join between edges with a distance of 2^{i-1} and edges with a distance that is less than or equal to 2^{i-1}. Unlike a linear strategy that requires a number of iterations that is equal to the longest path in the QCN, using a smart strategy, closure can be computed in just $O(\log n)$ rounds [2]. Example 3.6 shows the same example using a smart strategy.

Example 3.6: Using a smart strategy

```
Input QCN
(u,{bi},v,1)
(v,{bi},w,1)
(w,{bi},x,1)
(x,{bi},y,1)
(y,{bi},z,1)

Iteration 1
(u,{bi},v,1) ◇ (v,{bi},w,1) → (u,{bi},w,2)
(v,{bi},w,1) ◇ (w,{bi},x,1) → (v,{bi},x,2)
(w,{bi},x,1) ◇ (x,{bi},y,1) → (w,{bi},y,2)
(x,{bi},y,1) ◇ (y,{bi},z,1) → (x,{bi},z,2)

Iteration 2
(u,{bi},w,2) ◇ (w,{bi},x,1) → (u,{bi},x,3)
(v,{bi},x,2) ◇ (x,{bi},y,1) → (v,{bi},y,3)
(w,{bi},y,2) ◇ (y,{bi},z,1) → (w,{bi},z,3)
(u,{bi},w,2) ◇ (w,{bi},y,2) → (u,{bi},y,4)
(v,{bi},x,2) ◇ (x,{bi},z,2) → (v,{bi},z,4)

Iteration 3
(u,{bi},y,4) ◇ (y,{bi},z,1) → (u,{bi},z,5)
```

Using a smart strategy, comes at a cost, some iterations result in a large volume of derivations. Again, this can make the size of the join so large that the execution fails to complete. Furthermore,

implementing these strategies is complicated when computing $\diamond$-consistency, as we aren't simply interested in whether or not nodes can be reached, we need to know whether the newly inferred relation is stronger than the existing relation between nodes in the graph, and update the distance value accordingly. Section 3.3.3 describes how this is implemented in ParQR.

Another key point is that using a linear or smart strategy only prevents the reasoner making the same joins as a previous iteration, it doesn't prevent duplicate inferences within the same iteration. For example, in Figure 3.4 the same relation between u and z is inferred four times, each via a different path in the QCN. If the QCN is dense, this can create an explosion of relations, especially when using a smart strategy, that can again create performance issues for the reasoner.

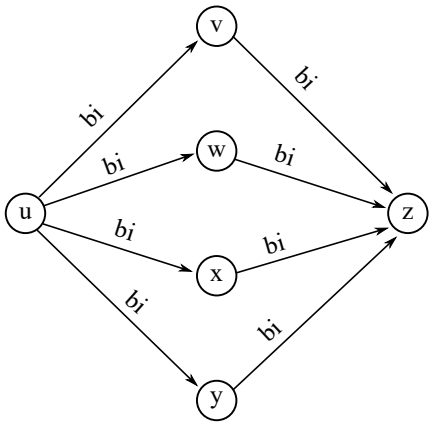


Figure 3.4: Duplicate inferences within an iteration

3.3 ParQR algorithms

The following section describes in detail the algorithms used by ParQR to perform QSTR. [1]

3.3.1 Main program execution

Algorithm 2 shows the main program execution for ParQR. The program requires two inputs, a QCN and a calculus. The QCN is a text file where lines represent edges in the network. Each line is split using standard string functions to generate tuples e.g. see the tuples in Example 3.5. ParQR is a generalised reasoner that is able to reason using any qualitative contraint calculi. In order to do this, it also needs a calculus as an input. The calculus is a simple text file, this time specifying the relations in a calculus, their inverses and the composition table for the calculus.

[1] Full code listings can be found at https://github.com/mmantle-hud/ParQR

Algorithm 2: ParQR - main program execution

```
1:  ParQR(QCN, calculus)
2:      QCN=QCN ∪ QCN^R
3:      possibleRels = generatePossibleRelations(QCN, calculus)
4:      compTbl = generateCompTbl(possibleRels, calculus)
5:      intersectionTbl = generateIntersectionTbl(possibleRels, calculus)
6:      QCN=consistency(QCN)
7:      count=0
8:      i=1
9:      while QCN.count() ≠ count
10:         count=QCN.count()
11:         newEdges=inference(QCN,i)
12:         QCN=consistency(QCN ∪ newEdges)
13:         i++
14:     end while
```

Generating the reverse of the network

Before reasoning over a QCN, it is first necessary to add the reverse of each edge to the network (line 2). This is needed to make sure that all possible derivations can be made. For example in Figure 3.2, if the initial QCN featured the edge $(y, \{EC\}, z)$ instead of $(z, \{EC\}, y)$, it wouldn't be possible to infer a relation between the variables x and y as inference works by matching head nodes with tail nodes. Therefore, to make sure that all possible inferences can be made, the reverse of the input QCN is generated as a pre-processing step.

Pre-computing composition and intersection

ParQR also pre-computes the results of the composition and intersection operations. Algorithm 2 refers to a *generatePossibleRelations()* function. This function looks up composition for all combinations of relations from the input QCN. The results of these composition operations can form the basis for additional compositions, so the process repeats until the results of all possible composition operations that could arise from the QCN have been computed. Once all possible relations have been computed, look-up tables for composition and intersection are generated.

This provides two performance benefits. First, as discussed in Chapter 2 if the composition involves disjunctive relations, it is necessary to look-up the composition for each pair of basic relations and take the union of these compositions. Typically this would be implemented as a nested loop. However, by pre-computing the results of composition, the operation is reduced to a single look-up, even for disjunctive relations. A second, and more important benefit, is that by pre-computing the results of both composition and intersection, it is then possible to store relations as integers, rather than as an array type structure. The memory demands made by the large join in the inference stage can partially be ameliorated by reducing the memory footprint of the datasets being joined. Using primitive data types such as integers reduces the memory consumption of the QCN allowing ParQR to handle larger joins.

Main program loop

After completing these pre-processing steps, the QCN is then tested for consistency (line 6) be-

fore entering the main loop of the program. The main *while* loop executes the inference stage followed by the consistency stage repeatedly until no more inferences are made and the closure is completed. Note that the *while* loop also keeps track of the iteration number, which is needed to determine which edges should be joined during the inference stage.

3.3.2 The inference stage

Algorithm 3: ParQR - inference (smart strategy)

```
1:  inference(QCN, i)
2:      //QCN: A collection of edges e.g. [(X,{TPP},Y,1), (Y,{TPP},Z,1), ...]
3:      //i: The iteration number e.g. 1

4:      headEdges=QCN
5:          .filter(edge ⇒ edge.distance = 2^(i−1))
6:          .map(edge ⇒ (edge.tailNode, edge)

7:      tailEdges = QCN
8:          .filter(edge ⇒ edge.distance ⩽ 2^(i−1))
9:          .map(edge ⇒ (edge.headNode, edge)

10:     joinedEdges = headEdges.join(tailEdges)

11:     newEdges=joinedEdges.map((key, (headEdge, tailEdge))⇒{
12:             headNode = headEdge.headNode
13:             tailNode = tailEdge.tailNode
14:             inferredRelation = lookUp(headEdge.relation, tailEdge.relation)
15:             distance = tailEdge.distance + headEdge.distance
16:             return (headNode, inferredRelation, tailNode, distance)
17:         })
18:         .filter(edge ⇒ edge.relation ≠ B)
19:
20:     return newEdges
```

Algorithm 3 shows the inference function using a smart strategy. Two sides of the join are generated in lines 4-6 and 7-9, with the iteration number being used to filter which edges participate in the join. A *map* operation is then used to iterate over the joined edges. Within this *map* operation the new relation is derived using the calculus's composition table, and the distance for the new edge is computed.

Many qualitative spatio-temporal reasoners maintain relations between all nodes in a network e.g. Algorithm 1, which leads to $O(n^2)$ memory requirements. This is one of the main reasons why many reasoners can't handle large scale networks consisting of hundreds of thousands, or millions of relations. ParQR doesn't maintain a complete network. It only stores edges where we know something about the relation between two variables i.e. their relation isn't the universal relation. In order to maintain this more streamlined QCN, ParQR filters out edges where composition results in the universal relation, see (line 18).

3.3.3 The consistency stage

Algorithm 4: ParQR - consistency (smart strategy)

```
1: consistency(QCN, i)
2:     //QCN: A collection of edges e.g. [(X,{TPP},Y,1), (Y,{TPP},Z,1), ...]
3:     //i: The iteration number e.g. 1

4:     keyedEdges = QCN.map(edge ⇒ (edge.tailInterval+'#'+edge.headInterval, edge))

5:     consistentEdges=keyedEdges.reduceByKey((edgeA,edgeB)⇒ {
6:             head = edgeA.headInterval
7:             tail = edgeA.tailInterval
8:             intersect = edgeA.relation ∩ edgeB.relation
9:             if |intersect| = 0
10:                //inconsistency detected
11:                stop()
12:            end if
13:            if edgeA.distance = edgeB.distance
14:                //they are the same it doesn't matter which we use
15:                distance = edgeA.distance
16:            else if edgeA.distance > edgeB.distance and |edgeB.relation| > |intersect|
17:                // the newly inferred relation (edgeA) is stronger
18:                distance = edgeA.distance
19:            else if edgeB.distance > edgeA.distance and |edgeA.relation| > |intersect|
20:                // the newly inferred relation (edgeB) is stronger
21:                distance = edgeB.distance
22:            else
23:                distance = Math.min(edgeA.distance, edgeB.distance)
24:            end if
25:            return (head, intersect, tail, distance)
26: }
27:    return consistentEdges
28:
```

The consistency function for the smart strategy is shown in Algorithm 4. The input to the consistency function is made up of the union of newly inferred edges and the complete QCN from the previous iteration, see Algorithm 2. The *consistency* function generates keys for all the edges by concatenating the value of the head node and the tail node. The *reduceByKey* operation then reduces all the edges between the same two nodes to a single edge. The *reduceByKey* function not only detects consistencies and removes duplicates, it also computes a distance value for the new edge. If the newly inferred relation is weaker than the existing relation between a pair of variables, there is no need to update the distance value of the edge. Any derivations made using this edge will simply be duplicates of previous derivations. However, if the new relation is stronger, it could form the basis for further derivations, in which case the distance value is updated to use the larger distance value. This edge can then be used in the left-hand side of the join in the next iteration.

3.3.4 Analysis

In parallel distributed applications, runtime is often dominated by communication costs and network bandwidth, as data is moved between different machines in a cluster. However, it is still

worth considering some aspects of the computational complexity of the above algorithms.

Main program execution The number of iterations executed by the *while* loop in Algorithm 2, the main program execution, has an upper bound dependent on the diameter of the input QCN, the specific strategy being employed, and the number of basic relations in a calculus (a constant factor). For example, for the smart strategy the time complexity is $\mathcal{O}(log(diam(QCN))$.

The inference stage The initial phase involves executing a *filter* operation and *map* operation over the entire input dataset twice, to generate the two sides of the join. Both the *filter* and *map* functions operate in linear time, and are trivial to parallelise. There are no dependencies between the input edges and the workload can be balanced evenly. In the worst case, the input to the inference stage is a complete graph, resulting in this initial filtering and mapping outputting $\mathcal{O}(n^2)$ edges.

This output is shuffled before being joined. From a time complexity point of view, shuffling is a linear time operation. However, in reality this is often much more expensive as data is moved around the cluster.

The exact join strategy used by Spark is determined by it's optimizer, it maybe be a broadcast join, hash join or sort-merge join. Parallelisation can ameliorate the costs of these joins, keys are distributed between partitions in the cluster, and the join is executed within a partition using a subset of the join keys. Unlike the initial *map* phase, depending on the distribution of the keys, the dataset may suffer some skew, with some join tasks taking longer to execute. In the case of a complete graph, assuming a sort-merge join, a local join task where a node is joined to all other nodes in the QCN will run in $\Omega(nlog(n) + nlog(n))$.

The join in this worst case scenario outputs $\mathcal{O}(n^3)$ joined edges. This is when each node is joined to all other nodes, using all join keys. See figure 3.4 for an illustration of how two nodes can be joined via many different paths. The final phase of the inference stage is a *map* operation which is used to infer the relation between two nodes, which again runs in linear time.

The consistency stage This involves a *map* operation with the input of size $\mathcal{O}(n^3)$ coming from inference stage. The map emits the same volume of data as it recieves which is shuffled and sorted for the *reduceByKey* operation. Again, following the worst case, where relations between two nodes have been derived using every other node as the join key, a single *reduceByKey* task has a time complexity of $\mathcal{O}(n)$. Following through the worst case, the entire *reduceByKey* phase can output no more than $\mathcal{O}(n^2)$ relations i.e. a complete graph, which then forms the input to the inference stage.

What is clear is that the bottleneck of the ParQR is the join operation at the inference stage. In reality, a complete graph doesn't form the input. Furthermore, this is very much a worst case, after the first iteration, the different join strategies ensure that the entire dataset doesn't participate in the join.

3.4 Related work

There are many reasoners that have implemented QSTR techniques to determine algebraic closure for QCNs. Related work can be divided into two categories, reasoners that take traditional approaches to QSTR, and reasoners designed specifically to reason over large scale qualitative constraint networks.

3.4.1 Traditional approaches to QSTR

Allen's original path consistency algorithm for Interval Algebra was discussed in section 2.1.5. Since the publication of Allen's IA paper several QSTR reasoners have been developed that use his original algorithm as the basis for determining $\diamond$-consistency for qualitative constraint networks. The most notable of these are Nebel's solver, [41], Renz's solver [51] and GQR [70].

These reasoners use a variety of optimisations to speed-up the computation of algebraic closure. These include different methods for computing composition, skipping techniques, and ordering of relations to determine which edges in a QCN should be processed first.

The basic method for computing composition involves using a nested loop to look-up the composition of basic relations using a $|\mathcal{B}| \cdot |\mathcal{B}|$ composition table. Similar to the pre-computation of composition and intersection described in section 3.3.1, various approaches have been used to speed-up this part of the reasoning process. For example, Renz's solver, which focusses only on RCC8, simply pre-computes all compositions to generate a 256 x 256 table. Composition then involves a single look-up. For more generalised reasoners this approach isn't always possible. For example, the RCC-23 calculus, which features 23 basic relations, would require a $2^{23} \cdot 2^{23}$ table which can prove impractical. Hogge's method is a middle ground where relations are pre-computed, but stored in four separate tables. Composition is then limited to four array accesses [9]. Such an approach is used in the GQR reasoner.

Many reasoners also skip some composition operations where it is known in advance that the result of composition will result in the universal relation e.g. if one of the relations involved is the universal relation, the result will also be the universal relation. Rules for specific calculi have also been implemented e.g. in the case of Interval Algebra, if one relation contains the basic relation *b(before)* and the other contains *bi(after)* there is no value in computing the result of composition as the result will be the universal relation.

The order in which edges are processed can effect reasoning performance. Some compositions result in strong, highly constrained relations, whereas other compositions result in weaker relations. Consider the following example adapted from van Beek [9]. If we first derive the relation between x and z using the following composition $x\{di\}y \diamond y\{d\}z \rightarrow x\{o, oi, s, si, d, di, f, fi, eq\}z$, this new relation is stronger than the universal relation, therefore $x\{o, oi, s, si, d, di, f, fi, eq\}z$ will be placed on the queue to be used as the basis for further inferences. If a relation between x and z is then derived using different edges e.g. $x\{fi\}w \diamond w\{d\}z \rightarrow x\{o, s, d\}z$, the relation between x and z will be updated again, and again placed on the queue. However, if the composition using

the edges *(x,w)* and *(w,z)* was done first, the *(x,z)* edge would only be placed on the queue once, as the second composition results in a weaker relation. Clearly reducing the number of times an edge is added to the queue will have an impact on reasoning runtimes. There are various heuristics that can be used to predict which edges will generate stronger relations e.g. a simple heuristic would simply give priority to edges featuring basic relations over disjunctive relations.

The current state of the art in terms of QSTR systems is GQR. This is a generalised reasoner that uses weight and cardinality heuristics, and can determine an appropriate strategy for pre-computing composition tables based on the input calculi [70].

A limitation of all the main traditional approaches to QSTR, including GQR, is that they maintain a representation of the complete network in memory. Typically implemented as a 2-dimensional array, this leads to $O(n^2)$ space requirements, see Algorithm 1. Reasoning over what might not appear to be an especially large network of 10,000 nodes would require the reasoner to store 100,000,000 relations between these nodes.

3.4.2 Reasoning with large scale qualitative constraint networks

In recent years there has been significant interest in reasoning over large scale qualitative constraint networks. Given the scalability limitations of traditional approaches to QSTR, a number of different techniques have been used to determining $\diamond$-consistency for QCNs.

Partial $\diamond$-consistency using chordal networks Beik and Hammond [12] first introduced the notion of partial algebraic closure. Partial algebraic closure is achieved by executing the following operations until a fixed point is reached.

$$\forall\{v_i, v_j\}, \{v_i, v_k\}, \{v_j, v_k\} \in E, C(v_i, v_j) \cap (C(v_i, v_j) \diamond C(v_j, v_k)) \rightarrow C(v_i, v_j)$$

This is very similar to the algebraic closure formula presented in section2.1.5. The key difference is that partial algebraic closure doesn't consider the relation between every pair of variables in a QCN, it only considers edges that are part of the input QCN. Consequently, algorithms for deciding partial algebraic closure are more efficient. They complete in $O(\delta \cdot |E| \cdot |\mathcal{B}|)$ time, where δ is the maximum degree of the graph, $|E|$ is the number of edges and $|\mathcal{B}|$ the number of basic relations in the calculus, see [12] for full details.

In the general case, partial algebraic closure isn't equivalent to algebraic closure. However, if a QCN is chordal then partial algebraic closure is sufficient for deciding $\diamond$-consistency.

A chordal network is one where any cycles that feature four or more vertices also feature an edge (a chord) between two of the vertices in the cycle. This results in a network where the holes have a size of three. Checking whether or not a graph is chordal can be done in $O(|V| + |E|)$ time. If a graph isn't chordal then it is possible to make it chordal by adding edges, known as fill edges, to the graph using a process known as triangulation. Again there are efficient algorithms that will add fill edges to a graph. However, minimal chordal completion i.e. adding the minimum number of fill edges necessary in order to make a graph chordal has been shown to be NP-Complete [73].

Sioutis and Koubarakis [62] consider computing algebraic closure for chordal networks. Using partial algebraic closure and chordal networks, they were able to limit the number of inferences made, and reason efficiently over large scale networks. Using a reasoner called Sarissa that uses these techniques, they were able to reason over an RCC8 network made up of 590,443 edges in 6 seconds [60]. Another key feature of Sarissa is that it doesn't maintain a copy of the complete network in memory, it only stores the edges of the chordal graph, so isn't limited by the $O(n^2)$ space requirements of traditional approaches to QSTR. The limitation of the triangulation approach is that it is only really suited to sparse network. Making dense networks chordal can often require many fill edges to be added to a network which results in an even denser network where the benefits of reasoning using partial algebraic closure become limited [60].

Partitioning QCNs An alternative approach to the problem of reasoning over large scale QCNs is to split a network into smaller parts, reason over each part separately and use these sub-network results to draw conclusions about the consistency of the complete network.

Doing this is possible for constraint calculi that have the *patchwork* property. For calculi with the patchwork property, if we have two $\diamond$-consistent QCNs, and these two QCNs are unified into a single larger network, as long as the unified QCNs agree on the relations between the variables they have in common, this larger network will also be $\diamond$-consistent [34] i.e. the two consistent networks can be *patched* together to create a larger also consistent network. There are many qualitative constraint calculi that display the *patchwork* property including the Point Algebra, and tractable subset of both RCC8($\widehat{\mathcal{H}}_8$) and IA($\mathcal{H}_{IA}$).

If partitioning is used, deciding $\diamond$-consistency for large scale networks then partially becomes a problem of efficiently partitioning a network. However, graph partitioning is an $\mathcal{NP}$-hard problem, with partitioning tools such as METIS [2] relying on approximation algorithms. The most notable reasoner to use network partitioning is the gp-rcc8 [45] reasoner. It delegates the partitioning to METIS and is then able to reason over RCC8 sub-networks using a parallel, but not distributed approach. The experimental evaluation of gp-rcc8 presented in [60] shows mixed results. gp-rcc8 is able to reason over large scale networks consisting of 5 million edges in 377 seconds. However, as noted in [60] this was only after some experimentation with partitioning parameters to generate suitably sized sub-networks. Furthermore, a significant proportion of this time was spent partitioning the network; such an approach is always going to be dependent on how easy the network is to partition.

Distributed approaches to large scale QSTR There are QSTR systems that, like ParQR, take a parallel distributed approach to reasoning. MRQUSAR [40] and MRQUTER[29] both use the MapReduce framework to implement distributed spatial(MRQUSAR) and temporal(MRQUTER) reasoners. QCNs consisting of 6 million edges were used in an evaluation of MRQUSAR showing that it can handle large scale networks. However, both reasoners show limitations. They both take

[2] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

quite a simple approach to determining algebraic closure e.g. they don't feature optimisations such as those presented in Section 3.2. Furthermore, they were evaluated using synthetically generated networks with attributes that posed few challenges for computing algebraic closure. For example, the evaluation for MRQUSAR used networks that don't have any cycles, and each vertex had a degree of one.

3.5 Evaluation

The performance of ParQR was been evaluated using a number of different experiments. The purpose of the evaluation was to find answers to the following questions:

- Can ParQR successfully reason over large scale datasets?
- Does the reasoning approach implemented in ParQR scale effectively?
- How does ParQR compare to existing state of the art reasoners?

In order to answer these questions a number of different datasets were used as the basis for the experiments, both synthetically generated QCNs and real world knowledge graphs.

3.5.1 Synthetically generated QCNs

Table 3.1: Synthetically generated interval algebra networks used in experiments 1-5

Experiment	No. of Nodes	No. of Edges	Ave. Degree	Ave. Label Size
1	30,000,000 - 150,000,000	30,000,000 - 150,000,000	2	1
2	6,000 - 30,000	30,000 - 150,000	10	1
3	100,000,000 - 500,000,000	100,000,000 - 500,000,000	2	6.4
4	2,000,000 - 10,000,000	10,000,000 - 50,000,000	10	6.4
5	30,000,000	30,000,000	2	1

Synthetically generated QCNs were needed to test the capabilities of the reasoner. By varying the numbers of nodes, edges and labels in a network it is possible create a wide range of QCNs that can be used to identify the strengths and limitations of the reasoner. Previous research into spatio-temporal reasoners used a variety of methods for generating QCNs that could be used for evaluation purposes. The approach taken to creating synthetic QCNs for this evaluation was based on the *H*-model [51] which in turn was based on the *S(n,p)* model [9]. The *H*-model accepts three parameters, the number of nodes, the average degree for nodes in the network, and the average label size. Label size refers to the cardinality of the relation e.g. the edge $x\{o, s, d\}z$ has a label size of three. Using these input parameters, Interval Algebra networks were generated. First intervals were created simply by randomly selecting points on a line. Then basic relations between these intervals were computed to obtain the desired average degree for the network.

Finally, if a label size greater than one was needed, the basic relations were replaced with disjunctive relations that were randomly selected from the $\mathcal{H}_{IA}$ subset. To ensure the resulting network would be consistent, these swaps were limited to disjunctive relations that contained the original basic relation. The characteristics of the synthetically generated QCNs used in the experiments are shown in Table 3.1. QCNs featuring an increasing number of nodes were used in experiments 1-4. Experiment 5 focussed on speed-up, scalability with respect to computing resources. Therefore a fixed QCN size was used, and this same QCN was reasoned over using different sized computing clusters.

3.5.2 Real world knowledge graphs

A number of large scale knowledge graphs have been used in previous experimental evaluations of qualitative spatio-temporal reasoners, see Table 3.2. As such they form somewhat of a benchmark that can be used to compare the performance of different reasoners. For example, the evaluations presented for gp-rcc8 [45] and Sarissa [60] use these knowledge graphs .

The knowledge graphs shown in Table 3.2 vary in size and degree, and at the time of writing represent the largest real world QCNs that reasoners have been asked to work with. All the QCNs are RCC8 networks.

Two of these QCNs, GADM1 and GADM2, feature inconsistencies. These inconsistencies are quickly identified during the first round of reasoning. Therefore, in order to fully test the capabilities of reasoners, these inconsistencies were removed. A second reason for removing the inconsistencies and checking for $\diamond$-consistency again is that there may be additional inconsistencies which can only be identified at later rounds of reasoning.

Table 3.2: Real world knowledge graphs used in experiment 6

	Nodes	Edges	Ave: Degree	Ave. Label Size
NUTS	2235	3176	2.84	1.99
ADM1	11761	44833	7.62	1
GADM1	42749	159600	7.46	1
GADM2	276727	590443	4.26	1.99
ADM2	1732999	5236270	6.04	1.98

3.5.3 Experiment setup

ParQR was written using the Scala[3] programming language and the Apache Spark distributed programming framework. Experiments were conducted using computing clusters provided by Google Dataproc[4], a cloud based service for running large scale data processing. For experiments 1-4 a computing cluster made up of 16 machines was used. Each machine was equipped with

[3]https://www.scala-lang.org/

[4]https://cloud.google.com/dataproc

8 virtual CPUs and 52 GB of memory. Experiment 5 was concerned with speed-up. For this experiment the same machine type was used (8 vCPUS and 52 GB of memory) but the number of machines in the computing cluster was varied. Clusters of sizes 2, 4, 8 and 16 were used. For all the experiments reasoning times were limited to 1 hour.

Experiment 6, reasoning using real world knowledge graphs, used the same 16 machine cluster used in experiments 1-4.

For the majority of experiments the smart strategy was used by ParQR. However, in some cases a linear strategy was used. If the linear strategy was used, this is clearly described in the analysis of results.

3.5.4 Results for synthetically generated knowledge graphs

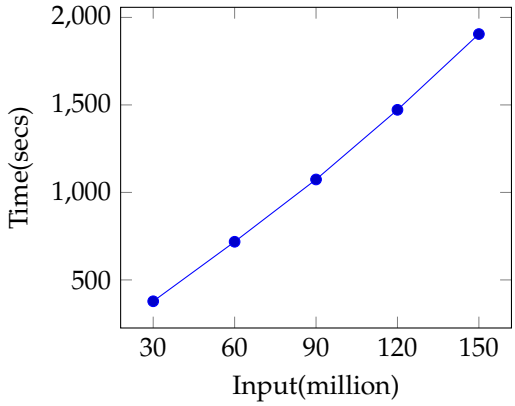


Figure 3.5: Experiment 1: Runtime as a function of input size on $\mathcal{H}_{IA}$ IA network instances $H(n,2,1)$

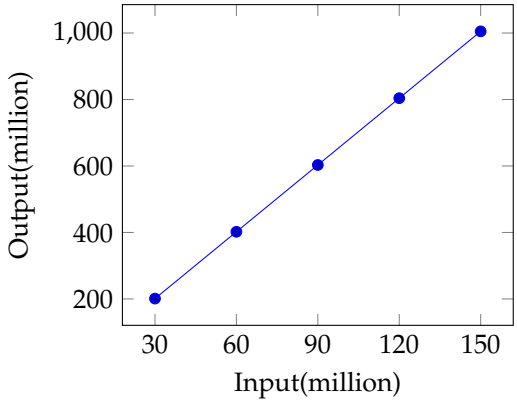


Figure 3.6: Experiment 1: Data volume output as a function of input size on $\mathcal{H}_{IA}$ IA network instances $H(n,2,1)$

Experiment 1

Figure 3.5 shows the runtime for Experiment 1 which involved reasoning using basic relations with an average degree of two. In the context of QSTR these are huge networks. The reasoner is able to handle networks with an input size of 150 million relations. For the largest of these QCNs, the reasoning results in an Interval Algebra network consisting of just over 1 billion relations, see Figure 3.6. Furthermore Figure 3.5 shows how the reasoner is able to scale effectively, linear regression analysis shows an R^2 value of 0.9974, runtime grows proportionally as the input size does.

Experiment 2

Experiment 2 involved the use of much denser networks. Like Experiment 1 basic relations were used when constructing the QCNs, but the average degree for each node was increased to ten. This presented a much greater challenge for ParQR. This can be seen in the runtimes in Figure 3.7

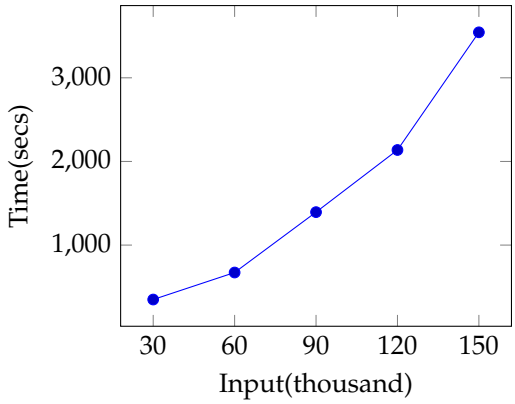


Figure 3.7: Experiment 2: Runtime as a function of input size on $\mathcal{H}_{IA}$ IA network instances $H(n,10,1)$

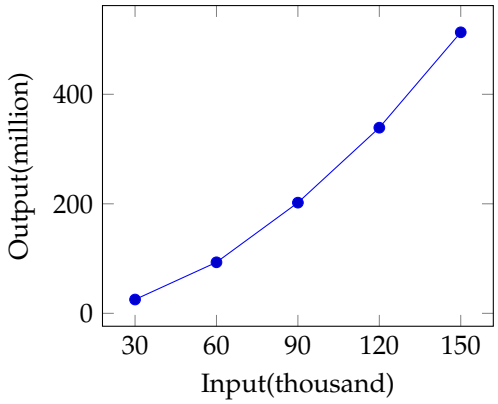


Figure 3.8: Experiment 2: Data volume output as a function of input size on $\mathcal{H}_{IA}$ IA network instances $H(n,10,1)$

which are longer than Experiment 1, even though much smaller networks have been used - thousands of edges, not millions. Moreover, the runtime no longer scales linearly as the input size increases, instead the runtime approaches quadratic growth. Regression analysis confirms this with an R^2 value of 0.9966. This can be explained by the reasoner output which also no longer grows linearly. The denser networks result in many more possible derivations e.g. reasoning using an input QCN of 150,000 edges results in final network size of 513,000,000 edges (see Figure 3.8). Even taking this into account, it may still seem unusual that the runtimes for Experiment 1 are significantly shorter. The total network sizes generated in Experiment 1 are larger, up to a billion edges so we might expect the Experiment 1 runtimes to be longer. However, the results can be explained by considering the analysis in Section 3.3.4; in dense networks the join can be very large and result in many relations between two nodes being derived within a single iteration. Many of these relations are subsequently filtered out, either because they are the universal relation, see Algorithm 3, or due to the *reduceByKey* phase in the consistency stage, so they don't show in the final dataset sizes. In fact, the join size was so large that it was necessary to use a linear join strategy rather than a smart strategy for Experiment 2. Although a linear strategy requires a greater number of iterations, the volume of data being processed within any given iteration is smaller, allowing the reasoner to cope with larger QCNs. Of course, the additional iterations add to the runtime, which also helps to explains the difference in execution time between Experiment 1 and Experiment 2.

Experiment 3 and Experiment 4

Experiment 3 was a repeat of Experiment 1 but the QCNs featured disjunctive relations. Figure 3.9 and Figure 3.10 show the results. Again, ParQR was able to handle very large scale knowledge graphs, this time consisting of 500 million edges. Like in Experiment 1, the runtimes and output QCN sizes scale linearly with respect to the input sizes. The runtimes for Experiments 3 are faster than Experiment 1, and this can be explained by the fact that fewer derivations are possible e.g. an

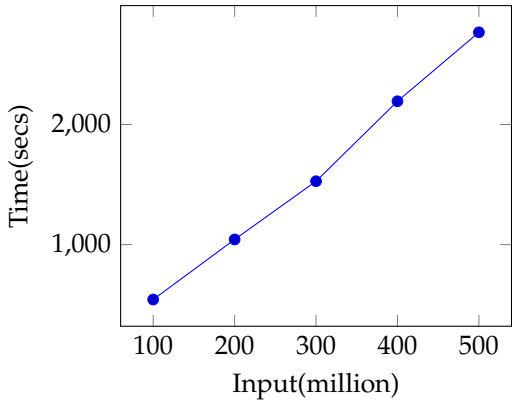


Figure 3.9: Experiment 3: Runtime as a function of input size on $\mathcal{H}_{IA}$ IA network instances $H(n,2,6.4)$

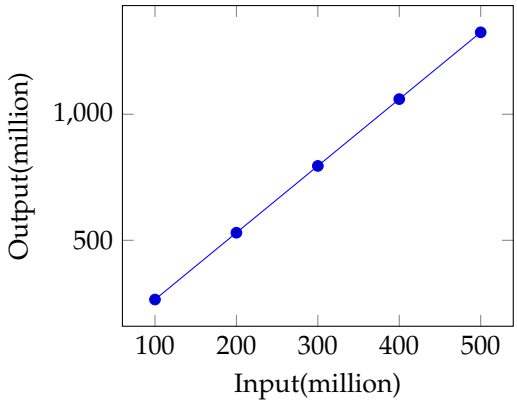


Figure 3.10: Experiment 3: Output as a function of input size on $\mathcal{H}_{IA}$ IA network instances $H(n,2,6.4)$

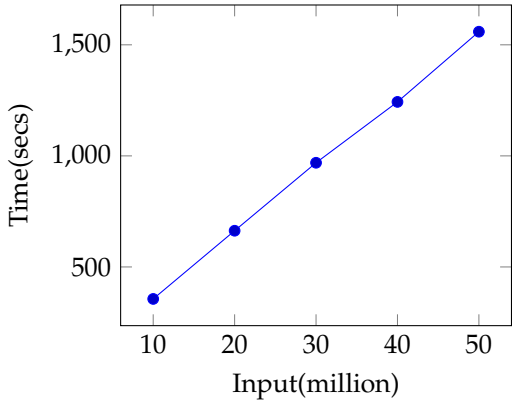


Figure 3.11: Experiment 4: Runtime as a function of input size on $\mathcal{H}_{IA}$ IA network instances $H(n,10,6.4)$

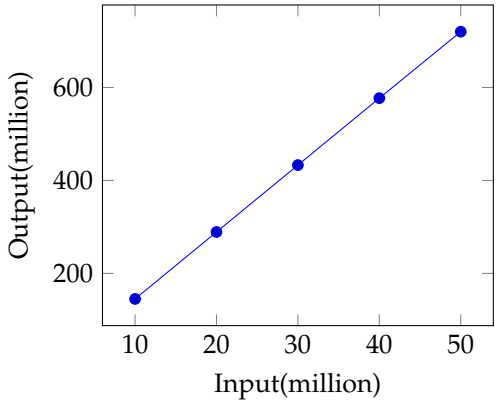


Figure 3.12: Experiment 4: Output as a function of input size on $\mathcal{H}_{IA}$ IA network instances $H(n,10,6.4)$

input QCN featuring 100 million edges results in an output consisting of 265 million edges. This is a consequence of using disjunctive relations. The results of composition for disjunctive relations are much more likely to result in the universal relation, which can't be used for further inferences, and then limits total the number of inferences that algebraic closure generates. Experiment 4 was a repeat of Experiment 2, but used disjunctive relations. Compared to Experiment 3, the denser knowledge graphs lead to a greater number of inferences being made, and result in slower runtimes, Figure 3.11 and Figure 3.12. However, unlike Experiment 2, the runtime and output still show linear growth. The presence of the disjunctive relations acts to limit the number of relations inferred by the reasoner meaning much large networks can be used than in Experiment 2.

Experiment 5

Figure 3.13 shows the results for Experiment 5 where scalability was considered in terms of

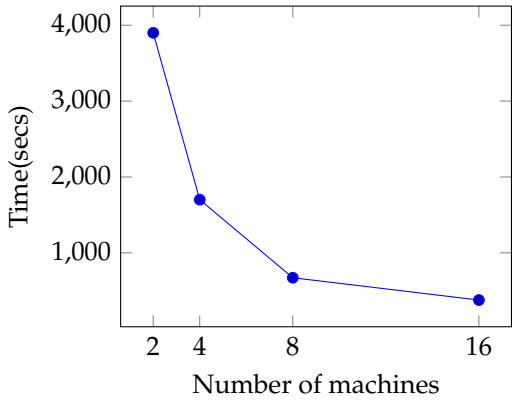


Figure 3.13: Experiment 5: Runtime as a function of number of machines in the computing cluster

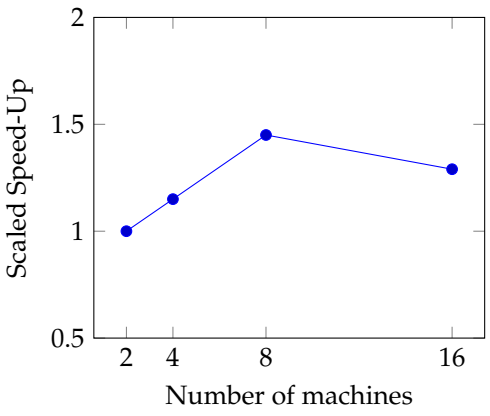


Figure 3.14: Experiment 5: Scaled speed-up

the cluster size used to execute ParQR's algorithms. As expected, runtime improves significantly as the number of machines used to process the knowledge graph are doubled. This indicates the algorithms are able to parallelise large parts of the processing effectively to speed up program execution. Figure 3.14 shows scaled speed-up. Scaled speed-up is a standard metric used to measure the performance of parallel systems and is specified as $\frac{t_1}{t_n} \div n$ where t_1 is the execution for a single node, n is the number of nodes in the cluster, and t_n the execution time for the n-node cluster. Because of the size of the QCN, a two node cluster was used as the baseline with the scaling factor being divided by two. Ideally we would like to see a scaled speed-up of at least one, when we double the number of machines in the cluster we half the reasoning times. Figure 3.14 shows super linear speed i.e. scaled speed-up greater than one, which again shows the algorithms allow for effective parallelisation. From 8 machines to 16 machines the speed-up slows. However, this is fairly typical for parallel distributed systems. At some point the benefits of further parallelisation for the specific dataset become limited, at the same time the overheads such as transferring data between different machines in the cluster increase with the number of machines.

3.5.5 Comparison with other reasoners

The performance of ParQR was compared to two existing reasoners, GQR and Sarissa, see Section 3.4 for details on these reasoners. GQR was chosen as it is widely considered to be the most sophisticated of the qualitative spatio-temporal reasoners that aren't designed specifically for working with large scale datasets. Sarissa was chosen as the research presented in [60] indicate it is the most capable of the reasoners that are designed specifically to tackle large scale spatio-temporal knowledge graphs. Neither the GQR or Sarissa reasoners are designed to work in a distributed environment. Therefore experiments for these reasoners were conducted using a single machine that had 8vCPUs and 52GB RAM.

	Size of Largest QCN (edges)	
Experiment	GQR	Sarissa
1	-	30,000,000
2	-	-
3	-	-
4	-	-

Table 3.3: Largest datasets reasoners could decide $\diamond$-consistency for

Synthetically generated IA networks

Experiments 1-4 were repeated using both GQR and Sarissa, the results are shown in Table 3.3 which shows the maximum network size that the reasoner could successfully handle. A dash indicates the reasoner was unable to compute $\diamond$-consistency for the smallest network used in the experiment. Neither reasoner was able to successfully reason over the large scale networks that ParQR could. The largest QCN that GQR could reason over was for Experiment 1 and consisted of 10,000 edges. As described in section 3.4.1 GQR stores a complete network in memory, resulting in $O(n^2)$ space requirements. Clearly, using such as approach, the memory limits of a single machine are unable to cope with large scale networks. Sarissa did return a result for Experiment 1. It was able to decide $\diamond$-consistency for a 30 million node network in 782 seconds. This is the smallest network used in Experiment 1. Even though Experiments 2 and 4 used smaller networks Sarissa failed to complete for these knowledge graphs. Experiments 2 and 4 use denser networks, which are less suited to the triangulation technique used in Sarissa.

	Runtime (seconds)		
Knowledge Graph	ParQR	GQR	Sarissa
NUTS	93 (smart strategy)	1.0	0.1
ADM1	211 (linear strategy)	-	395.6
GADM1	240 (linear strategy)	-	794.6
GADM2	476 (linear strategy)	-	5.8
ADM2	1702 (linear strategy)	-	662.8

Table 3.4: Runtime for computing $\diamond$-consistency for real world knowledge graphs (Experiment 6)

Real world knowledge graphs

Table 3.4 shows the runtime for computing $\diamond$-consistency for the real world knowledge graphs. The NUTS dataset is comparatively small in size, made up of just 3176 edges. Compared to Experiments 1-4, ParQR was fast, deciding $\diamond$-consistency in just 93 seconds. However, the distributed approach was much slower than both GQR and Sarissa that were able to decide consis-

tency rapidly, in 1.0 seconds and 0.1 seconds respectively. Consistent with the results in Table 3.3 this was the largest network GQR was able to successful reason over. The other knowledge graphs in Table 3.4 either failed to fit in memory or the runtime exceeded the 1 hour limit.

Both ParQR and Sarissa were able to successfully reason over all the networks. As expected Sarissa performed especially well on the most sparse of these networks, GADM2, completing in just 5.8 seconds. ParQR was able to outperform Sarissa for the denser networks (ADM1 and GADM1). For the sparse networks there is a big difference in runtime between ParQR and Sarissa. However, this isn't surprising. A distributed approach has inevitable overheads e.g. partitioning and distributing data to different machines in the network, and shuffling data between machines for operations such as *join* or *reduceByKey*. In an attempt to lessen the impact of these factors, ParQR was tested using the NUTS knowledge graph on a single machine. This resulted in a runtime of 55 secs. This is faster than the 16 machine cluster, but still considerably slower than GQR and Sarissa. Even without the network communication costs, the use of a distributed framework still places an additional burden on runtime e.g. data is still partitioned.

It is also worth comparing the runtimes for the real-world networks to the times recorded in Experiments 1-4. A knowledge graph such as ADM2 features $\approx$5 million edges and has an average degree of 6.04. It is denser than the knowledge graphs used in Experiment 1, but less dense than those used in Experiment 2. Even so, the runtime of 1702 seconds is longer than might be expected. The synthetically generated knowledge graphs, used in Experiment 1 were randomly generated leading to a uniform distribution of edges. A knowledge graph such as ADM2 has a structure that is scale-free [61] where the distribution of edges follows a power law resulting in a small number of nodes with a large number of edges. In a distributed environment this can lead to data skew. For example at the inference stage all the joins for a given node are processed by the same executor within the same partition, which can lead to uneven workloads between machines in the cluster. Even though tasks can run in parallel, the runtime is always bound by the time it takes for the longest running task to complete.

Although it wasn't needed for the datasets presented here there are strategies that can be used to cope with heavily skewed datasets. These include broadcasting, see Chapter 2, or salting keys to distribute the join more evenly.

Inflated graphs Sarissa was much faster that ParQR for the sparser, real world knowledge graphs. A final experiment was run that considered how large these knowledge graphs would need to be either for ParQR to decide consistency faster than Sarissa or for Sarissa to be unable to cope with the size of the network. In order to do this the original knowledge graphs were 'inflated'. The knowledge graphs were copied, with the nodes of the copied graph being renamed to avoid duplicates with the original. By generating multiple copies of the knowledge graph and then combining these copies into a single graph, it was possible to create successively larger QCNs that maintained similar characteristics to the original knowledge graph, and could be used to compare Sarissa and ParQR. Figure 3.15, Figure 3.16 and Figure 3.17 show the results of these

experiments. Sarissa was able to reason over all the knowledge graphs faster than ParQR until the knowledge graph reached a size where Sarissa was unable to complete. This is shown on the charts as a missing data point for Sarissa. For NUTS it was a QCN consisting of 25 million relations, for GADM2 the inflated graph was 30 million relations in size, and for ADM2 it was 31 million relations. These results and those of Experiments 1-4 indicate an upper limit of approximately 30 million relations for the size of knowledge graph that can be successfully handled by Sarissa.

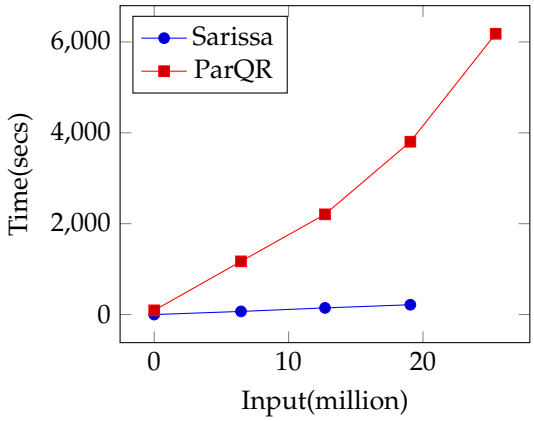


Figure 3.15: Runtime as a function of input size on scaled NUTS instances

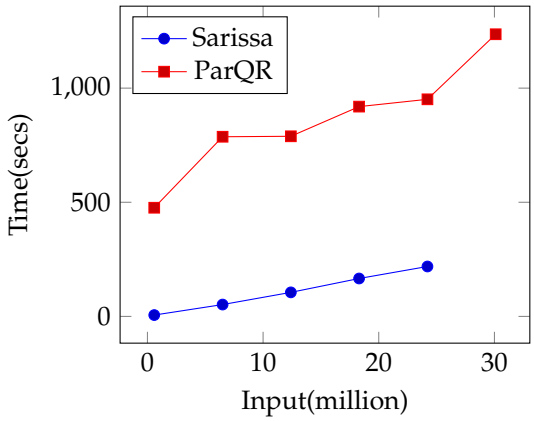


Figure 3.16: Runtime as a function of input size on scaled GADM2 instances

3.5.6 Conclusions

Summary Chapter 3 has presented ParQR, a distributed qualitative spatio-temporal reasoner. The reasoner uses novel techniques to implement $\diamond$-consistency algorithms in a distributed environment. It features several optimisation that allow it to deal with large scale knowledge graphs including:

- Efficient join strategies that prevent duplicate derivations.

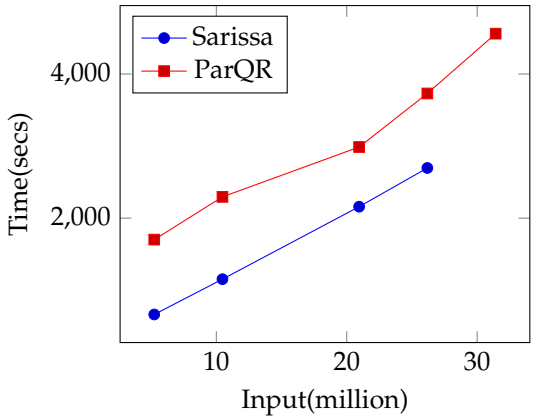


Figure 3.17: Runtime as a function of input size on scaled ADM2 instances

- Pre-computing calculi operations that speed-up processing and allow for streamlined data structures.

The reasoner has been shown to efficiently compute algebraic closure for qualitative constraint networks of very large sizes. For example, the experimental evaluation showed ParQR was able to reason over a synthetic dataset consisting of 150,000,000 edges. The evaluation presented in this chapter has compared ParQR to existing state of the art reasoners. Compared to traditional approaches to QSTR such as GQR, ParQR is able to handle QCNs of far greater size.

ParQR was also compared to Sarissa, a reasoner that is also designed to handle large scale networks. Although the results for the real world knowledge graphs showed that Sarissa was often able to compute algebraic closure faster than ParQR, Sarissa was unable to successfully handle all the knowledge graphs presented in the evaluation; a QCN consisting of 30 million edges was largest knowledge graph that Sarissa was able to handle.

Discussion Sarissa was able to outperform ParQR in a runtime comparison for many of the real world knowledge graphs. However, the distributed approach adopted in ParQR inevitably leads to longer runtimes so this isn't really a 'like for like' comparison. Furthermore, the evaluation has exposed two limitations of a reasoner such as Sarissa.

First, reasoners such as Sarissa tackle the problem of large scale QCNs by re-framing the problem as one of triangulation, similarly gp-rcc8 re-frames the problem as graph partitioning. As a consequence they are hampered by the limitations of the specific technique they use to avoid directly reasoning over large-scale networks. For example, Sarissa only has utility if the network is sparse and can be easily chordally completed. Similarly, gp-rcc8 is only effective if a suitable partitioning strategy can be found.

The second limitation is simply one of scale. Even if the characteristics of the knowledge graph make it suitable for partitioning or triangulation, there is still an upper limit in terms of the size of network that can be managed using a single machine approach. In the future the potential exists for there to be many more qualitative knowledge graphs of sizes even larger than those presented

in Table 3.2, for example IoT applications, reasoning over these will likely require a distributed approach such as that shown in ParQR.

Chapter 4

Enhanced spatial knowledge graph generation

Chapter 3 presented techniques for large scale distributed qualitative spatio-temporal reasoning. Chapters 4 and 5 of this thesis considers how QSTR techniques can be of use when querying large scale knowledge graphs. Chapter 5 describes ParQR-QE, a query engine that uses QSTR to answer spatial queries. However, before considering query answering, it is first necessary to consider the requirements for the type of knowledge graph that can be used as a basis for spatial querying and how such a knowledge graph can be created. This is the focus of this chapter, the creation of a large-scale enhanced knowledge graph capable of supporting a range of spatial queries.

4.1 Introduction

Many large scale semantic knowledge graphs such as YAGO [1] and DBpedia [2] have a spatial element to them. Spatial data is represented in these knowledge graphs in a number of ways. There are quantitative spatial triples where the object of the triple is a point geometry with latitude and longitude values. In YAGO these triples are defined by the *schema:geo* predicate e.g.

```
yago:Belgium    schema:geo    geo:50.641,4.668
```

There are also qualitative spatial triples. For example, in YAGO there are triples featuring the *schema:containedInPlace* predicate.

```
yago:Wellington_Museum,_Waterloo    schema:containedInPlace    yago:Waterloo,_Belgium
```

However, spatially querying such datasets is problematic for a number of reasons. To illustrate these problems a subset of the YAGO knowledge graph is shown in Table 4.1. The spatial elements of this knowledge graph are shown on a map in Figure 4.1.

[1] https://yago-knowledge.org/
[2] https://www.dbpedia.org/

subject	property	object
yago:Belgium	rdf:type	schema:Country
yago:Belgium	schema:geo	geo:50.641,4.668
yago:France	rdf:type	schema:Country
yago:France	schema:geo	geo:47,2
yago:Grand_Est	rdf:type	yago:Regions_of_France
yago:Grand_Est	schema:geo	geo:48.598,7.759
yago:Hauts-de-France	rdf:type	yago:Regions_of_France
yago:Hauts-de-France	schema:geo	geo:49.920,2.70
yago:Bastogne_War_Museum	rdf:type	schema:Museum
yago:Bastogne_War_Museum	schema:geo	geo:50.010,5.739
yago:Musée_Hergé	rdf:type	schema:Museum
yago:Waterloo,_Belgium	rdf:type	schema:Place
yago:Waterloo,_Belgium	schema:geo	geo:50.71, 4.38
yago:Wellington_Museum,_Waterloo	rdf:type	schema:Museum
yago:Wellington_Museum,_Waterloo	schema:containedInPlace	yago:Waterloo,_Belgium
yago:Arlon	schema:geo	geo:49.68,5.81
yago:Arlon	rdf:type	schema:Place

Table 4.1: Subset of the YAGO knowledge graph

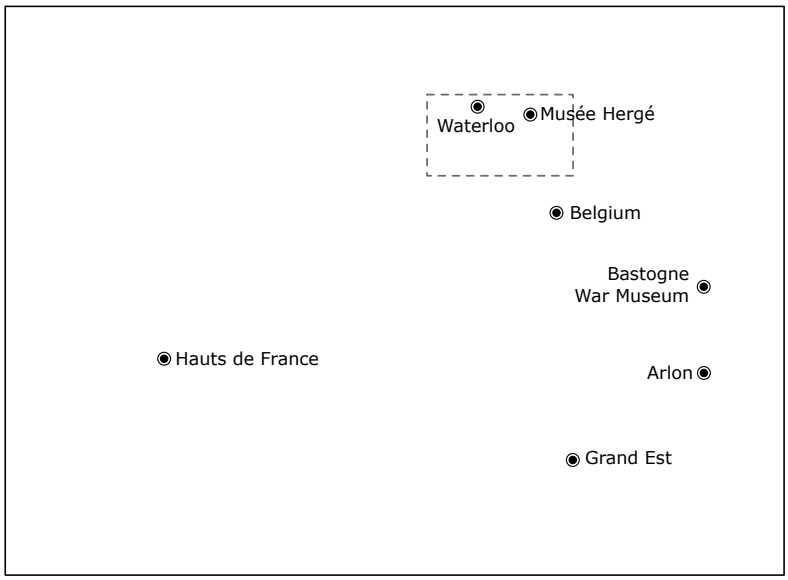


Figure 4.1: Map showing the location of point geometries from Table 4.1

Limitations of quantitative only reasoning

In traditional approaches to GIS, the focus is on quantitative data. However, large-scale knowledge graphs are typically composed of a variety of disparate data sources, both quantitative and

qualitative, with neither providing a comprehensive coverage of all spatial information. Consequently querying such a knowledge graph can lead to incomplete results. For example, consider a window query such as the GeoSPARQL query W1 (the query polygon is the dotted rectangle shown in Figure 4.1). Using quantitative querying techniques we could determine that the Hergé Museum is within the query window. However, if we refer to Table 4.1 we can see that The Wellington Museum has a *containedInPlace* relationship with Waterloo, and Waterloo is also within the query polygon. Using a purely quantitative approach, the Wellington Museum wouldn't be included in the results.

Query 4.3: Query W1

```
SELECT ?m
WHERE{
  ?m rdf:type schema:Museum.
  FILTER(geof:sfWithin(?m,"POLYGON((4.321423 50.5882119,4.6269803
    50.5869041,4.625607 50.7435868,4.3207364 50.7448903,4.321423
    50.5882119))"^^geo:wktLiteral))
}
```

What is needed is a hybrid approach, once the spatial entities within the query window have been identified, qualitative spatial reasoning is used to add in results from *containedInPlace* relations for these spatial entities.

Limitations of points

In semantic knowledge graphs such as DBpedia and Yago, geographic locations are approximated in the form of a single point (centroid). While this can often suffice for smaller spatial entities such as buildings or even towns and cities, representing large regions such as countries as a single set of coordinates can result in inaccurate query results, and limit the type of queries that are possible. Query C1 shows a containment query, *'find all the museums in Belgium'*. Even though all the museums in Table 4.1 are within Belgium, the point based representations mean it isn't possible to run queries of this type. Similarly, other types of query such as adjacency queries e.g. Query A3, *find the French regions that border Belgium*, aren't possible using point based approximations.

Query 4.4: Query C1

```
SELECT ?m
  WHERE {
  ?m rdf:type schema:Museum.
  ?m geo:sfWithin yago:Belgium.
}
```

Query 4.5: Query A3

```
SELECT ?r
  WHERE {
```

```
  ?r rdf:type yago:Regions_of_France.
  ?r geo:sfTouches yago:Belgium.
}
```

A possible solution would be to replace the simple point based approximations for large regions (countries, administrative areas) with full geometries i.e. polygons/multi-polygons. It would then be possible to answer a wider variety of spatial queries.

Problems of scale

Vector based datasets covering all countries and regions on the globe do exist e.g. GADM [3], OpenStreetMap [4]. There are also country specific datasets such as the UK's Ordnance Survey [5] dataset. However, replacing point geometries with these high resolution polygons also brings challenges. France's geometry in GADM is a multi-polygon consisting of over three hundred separate polygons and over 200,000 pairs of coordinates. Moreover, knowledge graphs such as YAGO and DBpedia can feature millions points. Geometric computation algorithms such as point-in-polygon tests and polygon adjacency tests aren't especially complex, but they do have time complexities that are dependent on the number of coordinates in the geometries being tested [54]. Executing spatial querying algorithms using a high volume of complex geometries can make query response times unfeasible. Furthermore, solely relying on such an approach still doesn't solve the problem of incomplete information in the knowledge graph, and the need to perform qualitative reasoning.

4.2 Requirements for an enhanced knowledge graph

In order to address these issues an enhanced knowledge graph is needed where:-

- Accurate, high resolution geometries i.e. polygons/multipolygons are integrated into the knowledge graph, and where appropriate these replace the simple point based approximations.
- Additional qualitative relations between spatial entities in the knowledge are computed e.g. EC relations between regions, containment relations between points and regions. This will then allow the knowledge graph to support qualitative spatial reasoning.

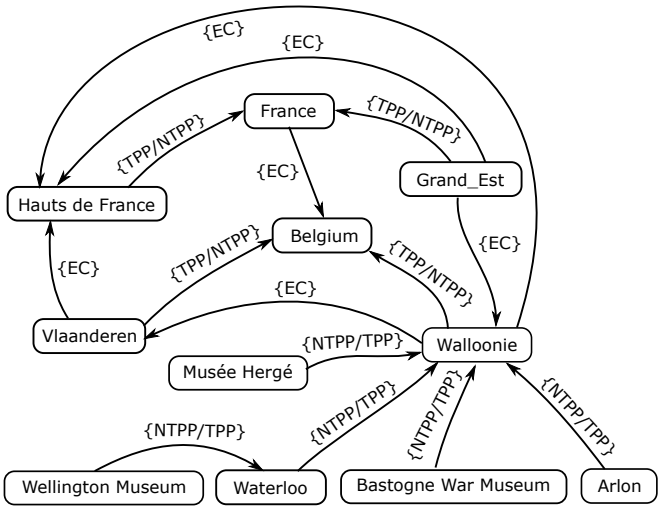
Figure 4.2 shows the knowledge graph from Table 4.1 with these enhancements. Figure 4.3 shows an updated map with polygons replacing the point based representations for countries and regions e.g. France, Grand Est, and with additional regions not present in the original knowledge graph e.g. Walloonie.

[3] https://gadm.org/

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

[5] https://osdatahub.os.uk/downloads/open

The enhanced graph consists of three elements. The geometries (either points or polygons) for spatial entities are stored in a geometries table, see Figure 4.2(c). An RCC8 network (Figure 4.2(a)) generated by computing relations between these entities and integrating existing qualitative spatial relations e.g. the *containedInPlace* relation is mapped to a {TPP,NTPP} relation. And the triples from the source knowledge graph vertically partitioned i.e. stored in two column tables, with a separate table for each RDF predicate, see Figure 4.2(b).



(a) RCC8 Network

subject	object
yago:Belgium	schema:Country
yago:Grand_Est	schema:Administrative_Area
yago:Musée_Hergé	schema:Museum
yago:Arlon	schema:Place
...	...

(b) rdf:type table

id	geomid	coords	geomtype
yago:Belgium	BEL#2	[[[...]]]	polygon
yago:France	FRA#23	[[[...]]]	polygon
yago:Musée_Hergé	yago:Musée_Hergé	[[[...]]]	point
yago:Hauts-de-France	FRA.7_1#1	[[[...]]]	polygon
...	...	...	...

(c) geometries table

Figure 4.2: Example enhanced knowledge graph

Using this enhanced knowledge graph, it is possible to execute a wider range of queries and obtain more complete results. Query W1 was a window query. Executing this query can now be done using a combination of quantitative and qualitative reasoning. First, the geometries of spatial entities 4.2(c) can be checked to see if they are contained by the query window. This would return Musée_Hergé and Waterloo (see Figure 4.3). Qualitative reasoning using these instances and the RCC8 network in Figure 4.2(a) can then be used to infer that the Wellington Museum should also be added to the results. These results can then be filtered using the *rdf:type* table to leave only the museums.

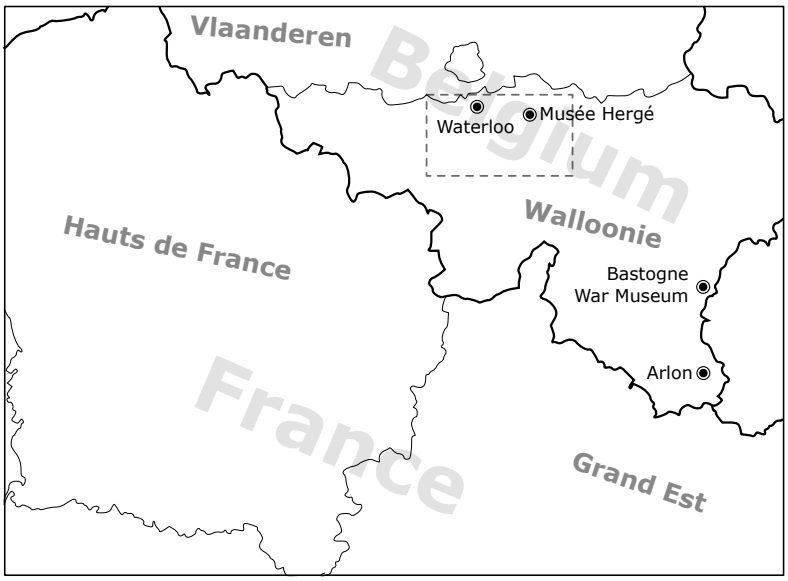


Figure 4.3: Updated map showing the regions as polygons

Using the more accurate geometries it is now possible to provide solutions to Queries C1 and A3. However, these queries can also be answered using purely qualitative methods, by reasoning over the RCC8 network. Without having to do any geometric computations, we can infer that all the museums in the knowledge graph are within Belgium. Similarly, for Query A3, we can infer that both Hauts-de-France and Grand Est are French regions that border Belgium. Moreover, because potentially complex geometries don't have to be checked, query answering solely using qualitative reasoning may also provide performance benefits, with queries executing in faster times than if a quantitative reasoning was used.

The remainder of this chapter describes the techniques used to integrate different datasets and generate an enhanced knowledge graph such as that shown in Figure 4.2.

4.3 Source Datasets

In order to create the knowledge graph two source datasets are needed, a large scale knowledge graph and a high quality spatial vector dataset. The example described here uses YAGO and GADM.

4.3.1 YAGO 4

YAGO 4 [6] is the latest version of the YAGO knowledge graph. The YAGO dataset was chosen because it is large enough to warrant a distributed approach, it has a focus on precision, and has a significant spatial dimension. As described in the introduction, spatial data is represented in the knowledge graph in both quantitative and qualitative form. Specifically the English Wikipedia

[6] https://yago-knowledge.org/

version of the knowledge graph was used [7]. This is made up of 209,591,985 triples, and includes 957,700 quantitative point-based *schema:geo* triples, and 1,109,652 *schema:containedInPlace* qualitative spatial triples.

The YAGO 4 knowledge graph was downloaded in the N-Triples format. Using Spark, the .nt files were parsed and stored in the Parquet format [8]. Parquet is a distributed columnar data storage format and was chosen for a number of reasons. Column based formats typically provide faster retrieval when a limited subset of attributes for particular objects are needed. Furthermore, depending on how the data has been partitioned, the Parquet format maintains statistics about blocks such as min/max values with the block. In the case of queries featuring a WHERE clause this can mean the entire table doesn't need to be scanned, some blocks can be skipped, which results in faster access times. The dataset was vertically partitioned by creating a separate table for each of the 151 predicates in the knowledge graph. For example, Table 4.2 shows part of a *schema:hasOccupation* table. Similar tables were generated for the other predicates in the knowledge graph.

subject	**object**
yago:Soe_Win_(prime_minister)	yago:Politician
yago:Chimaobi_Nwaogazi	yago:Football_player
yago:Finian_McGrath	yago:Politician
...	...

Table 4.2: Vertically partitioned *hasOccupation* table

An important consideration when querying large scale RDF datasets is the data layout of the storage. An obvious approach is to store the data as a single huge triples tables, made up of three columns (subject, property, object), and a row for each RDF triple. However, executing anything other than the simplest queries requires a self-join on this huge table. Each triple pattern in the query maps to an access of the triples tables, the results of which are joined together [1]. Single machine frameworks such as RDF-3X [43] take such an approach. However, by generating multiple indexes for the table, RDF-3X provides optimised data access for different query types, and fast query execution times are possible, specially if the predicates are selective. The triple table approach is less suited to a distributed environment. The triple table would need to be distributed, but doing so creates problems for indexing, where global indexes are difficult to implement and maintain in a distributed environment [56]. An alternative is the property table approach, where similar instances are grouped together in separate tables. The columns for these tables are the properties these entities have in common e.g. a *BookType* table might have rows for each instance of the type *Book*, and columns for the title, author, publication date etc. This layout can work effectively for star-shaped queries i.e. where we need to retrieve multiple properties for an instance, as this simply requires filtering on a single table. However, other query shapes e.g. linear

[7] https://yago-knowledge.org/data/yago4/en/
[8] https://parquet.apache.org/

queries are more problematic. Furthermore, the graph like nature of RDF means not all instances of the same type will have all the properties defined in the table, leading to lots of NULL values. Also multi-valued properties, just like in the relational model, can't easily be accommodated by a property table approach [1].

The approach adopted here is to use vertical partitioning (VP). Vertical partitioning splits knowledge graph into separate tables, one table for each property. The resulting tables each have two columns, for the subject and object linked by the property. Such an approach addresses many of the issues related to property tables. For example, there is no need for NULL values when subjects of a certain type don't have a particular property. Subjects with multiple values for a property can be easily accommodated as additional rows in the property table. However, the biggest advantage in a distributed environment, is that VP can significantly limit the volume of data that needs to be read. Only tables whose properties are referenced in the query need to be read. This reduction in input leads to to faster query times [56].

4.3.2 GADM

The Database of Global Administrative Areas (GADM) is a high resolution, vector based dataset of countries and their administrative areas. The most recent version of the GADM dataset, GADM 3.6, was used. The dataset covers the whole of the Earth. The data is organised in layers, for example layer 0 contains the boundaries of countries, layer 1 the first level administrative areas within countries. Depending on the country there may be further subdivisions e.g. French regions are divided into departments (level 2), and then arrondissements (level 3), cantons (level 4) and communes (level 5). The work in this thesis only uses layer 0 and layer 1. Further layers could be used if needed, but by using the first two layers, we get a knowledge graph that is rich enough to allow for QSTR and large enough to require a distributed approach to processing. In layer 0 there are 256 different countries, and in layer 1 there are 3610 administrative regions.

On the face of it, the dataset doesn't appear especially challenging to deal with. However, the dataset has a high resolution, with many of the geometries being large and complex. For example, Canada is a multi-polygon made up of 24,481 polygons and 3,889,947 vertices. The size of these geometries proved to be the biggest obstacle when processing the dataset. Distributed computing frameworks such as Spark are designed to deal with a high volume of records that have simple data structures. These simple records can be easily shuffled and joined, and the high volume allows for effective parallelisation. In comparison the GADM dataset is made up of a small number of large records with a complex, nested data structure for the region geometry. Executing operations such as joins using such a dataset can be difficult using distributed frameworks such as Spark.

GADM data is available as shapefile [9]. Using MapShaper [10], layer 0 and layer 1 were extracted in GeoJSON format. This formed the input to Spark. The GeoJSON data for each layer was parsed

[9] https://gadm.org/download_world.html
[10] https://mapshaper.org/

and, like the YAGO data, stored in Parquet format, see Table 4.3.

Multi-polygon regions were split into separate polygons e.g. the United Kingdom is a multi-polygon made up of 920 separate polygons, each polygon was stored as a separate row. This resulted in 116,995 polygons for layer 0 and 117,891 polygons for layer 1. For many operations, the simpler polygon representation will suffice. For example, consider attempting to determine if a specific administrative area has an TPP relationship with its parent country. If a spatial index indicates that only one of the country's polygons possibly contains the administrative area, it isn't necessary to check against all the different parts of a country, which will clearly speed up geometric computation. Furthermore, dealing with a higher volume of simpler data structures, that can be operated on in parallel, makes the dataset more suited to distributed processing.

regionid	**polygonid**	**name**	**coords**	**partid**	**polycount**
BEL.2_1	BEL.2_1#1	Vlaanderen	[[[50.7603,5.692...	1	4
BEL.2_1	BEL.2_1#2	Vlaanderen	[[[51.4544,4.963...	2	4
BEL.3_1	BEL.3_1#2	Wallonie	[[[50.75196,3.18...	2	2
FRA.6_1	FRA.6_1#1	Grand-Est	[[[48.39027,3.41...	1	1
FRA.7_1	FRA.7_1#1	Hauts-de-France	[[[51.05625,2.34...	1	2
FRA.7_1	FRA.7_1#2	Hauts-de-France	[[[48.85191,3.48...	2	2
...	...	...	...	...	...

Table 4.3: GADM polygon table

The data structure presented in Table 4.3 forms the basis for many of the tasks that are performed when generating and querying the enhanced knowledge graph so it is worth noting some features of this table:

- Each administrative area has a unique identifier, a *regionid*, which comes directly from the original GADM dataset and is based on the ISO 3166 standard e.g. FRA for France. These are hierarchical, for example, BEL.2_1 indicates a level 1 administrative area.

- As described above, multi-polygons are stored as separate polygons. Consequently an additional unique identifier, *polygonid*, was generated for each polygon based on the *regionid* and a simple numbering of the polygons for that region e.g. BEL.2_1#3 is the third polygon that makes up the region with a *regionid* of BEL.2_1. It was also necessary to maintain the number of polygons that make up the region (*polycount*). For some operations and queries e.g. is a region within a query window, it is necessary to know whether all the polygons that make up the region lie within the window.

- The geometries were stored in the *coords* column as a nested array in the form *[rings [coordinates[double]]]*. Polygons have an outer ring and optionally inner rings (holes). Alternative representations were considered e.g. WKT and serialising an instance of Polygon type objects. It was found that the most effective approach was to store the raw coordinates as

arrays and then instantiate instances of spatial objects e.g. Points, Polygons etc. when geometric operations needed to be performed.

In addition to storing the full geometries, it was also necessary to store the minimal bounding boxes (mbbs) for regions. A minimal bounding box is the smallest rectangle that contains all the points of a geometry. An mbb serves as a lightweight approximation of the full geometry. These were stored in a separate table, at the region level i.e. a single mbb for each region, not each polygon.

4.4 Creating the knowledge graph

As described above using these datasets as a starting point an enhanced knowledge graph needed to be generated that could support GeoSPARQL queries and qualitative spatial reasoning. Specifically, the following tasks needed to be performed:-

1. Compute RCC8 relations between spatial entities.
    - Compute EC relations between regions.
    - Compute containment relations between points and regions.
2. Match regions from GADM with points from YAGO e.g. *BEL* from GADM matches with *yago:Belgium*. Spatial queries can then use the full geometry for a region instead of the simple point based representation.

A note on the generality of the algorithms Although a specific case explored here uses YAGO and GADM, it is important to note that once the input datasets have been parsed, the algorithms described in this chapter [11] could work with any dataset. For example, DBpedia [12] instead of YAGO or Natural Earth [13] for the region polygon data. GADM has been chosen because it provides comprehensive coverage of the planet at high resolution and therefore is reflective of the most challenging dataset the algorithms presented here are likely to face.

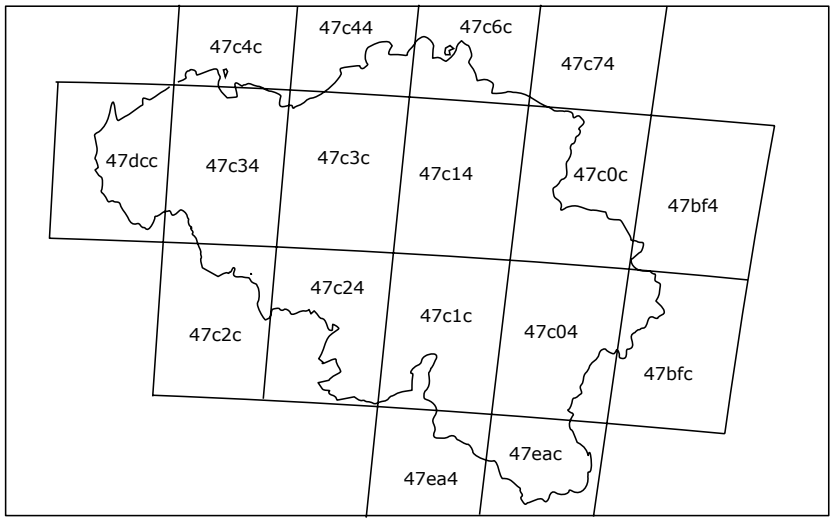
4.4.1 Spatial Indexing

Before describing the specifics of each of these tasks it is worth presenting the spatial indexing approach that forms the basis for many of these tasks and the query engine presented in Chapter 5. Completing the tasks outlined above has considerable challenges. For example, consider computing EC relations between regions in the GADM dataset. Essentially this is a huge spatial join, where for all regions we need to find all other regions that share a border. A naive approach would involve testing the geometry of every region with other region i.e. taking the Cartesian product of region geometries and testing for adjacency. This isn't feasible due to the volume of

[11] Full code listings can be found at https://github.com/mmantle-hud/KG-Generator
[12] https://www.dbpedia.org/
[13] https://www.naturalearthdata.com/

data that is generated in the join, and the need to execute a large number of complex geometric computations.

Instead of using the full geometries for spatial entities, approximations of the geometries can be used which significantly reduces the volume of data that needs to be joined and processed. As such these approximations form a spatial index, where using this lightweight representation, initial filtering is performed to remove regions that definitely won't share a border, leaving only candidate objects where there is a possibility of them touching. Finally, with a restricted set of spatial objects, a much smaller set of full geometries are accessed and tested to actually determine which regions touch.



cellid	regionid	polygonid
47c4c	BEL	BEL#1
47c6c	BEL	BEL#2
47c4c	BEL	BEL#3
47dcc	BEL	BEL#3
47c34	BEL	BEL#3
47c2c	BEL	BEL#3
...	...	...

(a)Cell coverings

cellid	regionid	polygonid	coords
47c4	BEL	BEL#1	[[[51.36, 3.37...
47c4	BEL	BEL#2	[[[51.42, 4.945...
47c4	BEL	BEL#3	[[[49.49, 5.470...
47dcc	BEL	BEL#3	[[[51.06, 2.58...
47c34	BEL	BEL#3	[[[51.21, 3.27...
47c2c	BEL	BEL#3	[[[50.55, 3.39...
47c04	BEL	BEL#3	[[[50.35, 5.35...
...	...	...	...

(b) Polygon-cell intersections

Figure 4.4: Cell coverings at S2 level 7 for Belgium

The index used in processing the spatial data for the generation of the knowledge graph is based on a grid. By splitting the surface of the planet using a grid, regions can be approximated using cells in this grid. The S2Geometry [14] was used to provide these spatial approximations. The S2Geometry library splits the Earth's surface into a hierarchy of cells arranged in different levels. For example at level 0 the Earth is divided into 6 cells, thousands of kilometres in width, at level 19 there are billions of cells approx 20m in width. Each cell has a unique, hierarchically

[14]https://s2geometry.io/

numbered cell id. The library provides classes that will generate cell ids given a valid geometric object e.g. a point or polygon. For example, Figure 4.4 shows a collection of S2 cells at level 7 that cover Belgium. Using S2Geometry a number of separate spatial indexes were generated.

S2 cell coverings for GADM polygons. For each polygon in GADM layer 0 and layer 1, a list of cells that cover the polygon were generated and stored, e.g. Figure 4.4(a). Importantly, the geometries of the region polygons aren't stored, only the cell ids, as a result the data in this table can be easily joined and shuffled in parallel, distributed processing.

S2 cells for YAGO points YAGO points were indexed in a similar way. The latitude and longitude values were extracted from triples featuring the *schema:geo* predicate. The S2 geometry library was used to identify which cell each point resides in. The cell id, along with the YAGO id, and the coordinates were saved, see Table 4.4.

cellid	id	coords
47c1c	yago:Sombreffe_Castle	[50.53, 4.59]
47c3c	yago:Waterloo,_Belgium	[50.71, 4.38]
47c14	yago:Musée_Hergé	[50.66, 4.61]
47c3c	yago:Flanders_Expo	[51.02, 3.69]
...	...	...

Table 4.4: Cell ids for YAGO points

Polygon-cell intersections Finally, the intersections between GADM polygons and S2 cells were computed. Using the intersection was beneficial when executing some geometric computations. For example, consider a point query where we need to identify which region contains a specified point. Without the use of polygon-cell intersections this query would involve joining the table in Figure 4.4(a) with Table 4.4 on the basis of *cellid* to find candidate polygons that might contain the point. The geometries of these candidate polygons would then need to be checked by joining with Table 4.3 to find which geometry contains the point. This can be sped up if the simpler geometry of the intersection between the region polygon and the cell is used instead. In some cases e.g. when a cell is completely contained by a region, the check can be reduced to a polygon that consists of the four vertices of the cell. Figure 4.4(b) shows example polygon/cell intersections for Belgium.

The algorithm for generating the cell ids and intersections for the polygons is described in Algorithm 5. The algorithm has two parts, generating a cell covering for each polygon, and computing the intersection between the region polygon and S2 cell. The first part is largely handled by the S2 library. *convertToS2Polygon()* is a custom function that simply instantiates an instance of an *S2Polygon* object by iterating through the rings and coordinates of a region's geometry. The process is reasonably involved, and not especially interesting, so isn't shown. The *S2RegionCoverer*

object comes from the S2Geometry library and is used to generate the cells that cover a region. Multiple cells are generated for each region, so this array of cells is flattened using *flatMap*.

Algorithm 5: Generate cell coverings

```
1:  generateCellCoverings(polygons, s2Level, generateIntersections) {
2:      //polygons: see Table 4.3
3:      //s2Level: the level to generate cells for e.g. 7
4:      //generateIntersections: Boolean value, specifying whether or not intersections are needed

5:      //generate the cells for each polygon
6:      cellCoverings = polygons.flatMap(polygon⇒ {
7:          s2Polygon = convertToS2Polygon(polygon.coords)
8:          regionCoverer = new S2RegionCoverer()
9:          regionCoverer.setMaxLevel(s2level)
10:         regionCoverer.setMinLevel(s2level)
11:         cellids = regionCoverer.getCovering(s2Polygon)
12:         return cellids.map(cellid⇒(cellid, polygon.regionid, polygon.polygonid)
13:     })

14:     cellCoverings.save() // see Figure 4.4(a)

15:     if(generateIntersections) {
16:         //broadcast the region polygons
17:         polygonsB = spark.broadcast(polygons)

18:         //generate polygon cell intersections
19:         polygonCellIntersections = cellCoverings.map(polygonCellId⇒ {
20:             //look-up the polygon using the broadcast variable
21:             polygon = polygonsB.value(polygonCellId.polygonid)
22:             //get the coordinates of the S2 cell
23:             cellCoords= getCoordsFromCellId(polygonCellId.cellid)
24:             //compute the intersection between the polygon and S2 cell
25:             intersection = getPolygonCellIntersection(polygon.coords, cellCoords)
26:             return (polygonCellId.cellid, polygonCellId.regionid, polygonCellId.polygonid, intersection)
27:         })
28:         polygonCellIntersections.save() //see Figure 4.4(b)
29:     }
30: }
```

In order to generate the intersections a broadcast (or map-side join) strategy is used. A copy of the polygon dataset is sent to each node in the computing cluster (line 17). The intersections are then computed by looking up the coordinates for the polygon using this broadcast variable. The broadcast strategy was used to avoid having to perform a large join. Multiple cell ids are generated for each polygon, using a default join would involve duplicating the polygon data for each cell the polygon overlaps. Given the size of the polygon geometries, this quickly becomes a significant bottleneck. Using a broadcast strategy makes the join more manageable. A broadcast strategy does have limitations, mainly that the dataset needs to fit in memory. For this reason, computing cell coverings and intersections for polygons was executed one layer at a time.

Throughout the generation of the enhanced knowledge graph two different spatial processing libraries were used. The S2Geometry library was used to provide the grid and perform related functions such as identifying which cells overlap polygons. The second library used was the ESRI

Geometry API [15]. This was used to provide fundamental topological and relational operations e.g. intersections, touches, contains etc. The actual computation takes places in the *getCellPolygonIntersection()* function, where the *Intersection* operation from the ESRI library was used to compute the intersection. Again, for clarity, the full details aren't shown. However, Algorithm 7 shows a similar function for testing adjacency.

A variety of approximations were generated using Algorithm 5. Cell coverings at an S2 level of 9 were generated for GADM layer 0 and layer 1. These provide an accurate approximation of regions using a large number of cells per regions. Polygon-cell intersections at an S2 level of 5 were also generated for both layers. A lower level was used as this sped up the computation of intersections, and was sufficient for the tasks where intersections were needed.

Analysis Although Algorithm 5 is heavily reliant on library code (to compute the cell coverings for polygons (S2Geometry), and cell polygon intersections (ESRI)), it is still worth considering some aspects of the algorithm's computational complexity. The initial *flatMap* operation runs in $\mathcal{O}(n)$ time where n is the number of input polygons. This can be parallelised trivially to significantly speed-up execution time. The outputs from this operation are multiple tuples for each polygon, one for each each S2 cell the polygon overlaps. The number of cells is dependent on the S2 level. At each increasing S2 level cells are split into 4 smaller cells. Therefore this *flatMap* operation emits $\mathcal{O}(n4^k)$ tuples, where k is the S2 level.

The second function *filter* implements a broadcast join, which runs in $\mathcal{O}(m+n)$ time where m is the numbers of cell tuples output from the previous *flatMap* operation and n the number of polygons. Again this operation can be parallelised, with the cell tuples being distributed across machines in the cluster.

4.4.2 Computing EC relations between regions

Once these spatial approximations have been created the tasks outlined at the start of section 4.4 could be completed. Algorithm 6 shows the computation of EC relations between different regions.

[15] https://github.com/Esri/geometry-api-java

Algorithm 6: Computing EC relations between regions

```
computeEC(cellCoverings, polygons) {
    //cellCoverings:see Figure 4.4(a)
    //polygons:see Table 4.3

    //broadcast the region polygons
    polygonsB = spark.broadcast(polygons)

    //join using cell id to identify candidates
    cellCoveringsL = cellCoverings
    cellCoveringsR = cellCoverings
    candidates = cellCoveringsL .join(cellCoveringsR,
        cellCoveringsL.cellid = cellCoveringsR.cellid &&
        cellCoveringsL.regionid ≠ cellCoveringsR.regionid && //no joins to self
        cellCoveringsL.polygonid < cellCoveringsR.polygonid //prevent duplicates
        )

    //for each candidate pair, look up the geometries and test if they touch
    adjacentPolygons = candidates.filter(candidatePair ⇒ {
        // get the polygon ids from the candidate pair
        polygonid1 = candidatePair._1.polygonid
        polygonid2 = candidatePair._2.polygonid
        //look-up the polygon using the broadcast variable
        polygon1 = polygonsB.value(polygonid1)
        polygon2 = polygonsB.value(polygonid2)
        //test if the polygons touch
        return touches(polygon1.coords,polygon2.coords)
    })

    //drop the polygon ids to only leave the regionids
    ecRelations = adjacentPolygons.map(adjacentPair ⇒ {
        return (adjacentPair.regionid1, "EC", adjacentPair.regionid2)
    }).distinct()

    //save the EC relations
    ecRelations.save() e.g. (FRA.7_1, "EC", BEL.2_1)
}
```

Algorithm 6 was executed a layer at a time i.e. EC relations between different countries, and between different level 1 regions were computed, but not between countries and level 1 regions. This is because spatial reasoning can be used to derive the EC relation between different layers at query time e.g.

$$BEL\{EC\}FRA \diamond FRA\{TPPi, NTPPi\}FRA.7\_1 \rightarrow BEL\{EC, DC\}FRA.7\_1$$

$$BEL\{TPPi, NTPPi\}BEL.2\_1 \diamond BEL.2\_1\{EC\}FRA.7\_1 \rightarrow BEL\{EC, PO, TPPi, NTPPi\}FRA.7\_1$$

$$BEL\{EC, DC\}FRA.7\_1 \cap BEL\{EC, PO, TPPi, NTPPi\}FRA.7\_1 \rightarrow BEL\{EC\}FRA.7\_1$$

Furthermore, as described previously, processing using these large scale geometries has challenges, using a layer at a time makes the computation more manageable. Even using a layer at a time, this is still a large scale spatial join. The program features a number of design choices that make the computation possible. The initial, potentially expensive, join operation is done using

the cell coverings for polygons. This lightweight representation doesn't feature any coordinates, and doesn't present any performance problems for Spark as each of these records is a simple tuple made up of three strings. Computing EC relations also makes use of a broadcast variable to implement a map-side join. Once candidate polygons have been identified through the initial cell coverings join, the actual geometries are looked up using this local copy of the polygon geometries. The actual checking of the geometries was performed using the ESRI API and the *Touches* operation. Algorithm 6 refers to a *touches* function, this is simply a wrapper function that creates instances of ESRI Polygon objects, and invokes the *Touches* operation from the ESRI library, see Algorithm 7.

Algorithm 7: The *touches* function

```
1: touches(coords1, coords2) {
2:     // coords1 Polygon coordinates: Array[Array[Array[Double]]],
3:     // coords2 Polygon coordinates: Array[Array[Array[Double]]]
4:     geometry1 = convertToESRIPolygon(coords1)
5:     geometry2 = convertToESRIPolygon(coords2)
6:     return OperatorTouches.local().execute(geometry1, geometry2,null, null)
7: }
```

Algorithm 6 identifies polygons that touch. Eventually, when answering queries, we are only interested in relations between (multi-polygon) regions, not individual polygons, so the final step of the algorithm involves dropping the polygonids to leave the regionids.

Analysis Algorithm 6 involves three stages. As with previous analyses, we assume a sort merge join for Spark's default join strategy, so the initial join between cell coverings will execute in $\mathcal{O}(nlogn)$. A second join is then implemented in the *filter* operation using a broadcast strategy. This will run with time complexity of $\mathcal{O}(m+n)$. The final operation involves *map* and *distinct* operations. As always the *map* runs in linear timer and is an embarrassingly parallel operation. The *distinct* is more complex as it requires data to be sorted and shuffled, and will run in $\mathcal{O}(nlogn)$ time. As already stated, runtime in distributed parallel programs is dominated by IO operations, and the size of the objects being processed. The first *join* and final *map* and *distinct* operations involve simple tuples, so in practice they execute quickly. Although the *filter* operation has linear time complexity, this dominates running time as it involves using the full polygon geometries.

4.4.3 Computing containment relations between points and regions

RCC8 relations between regions and points also need to be computed. The approach taken in generating these containment relations is shown in Algorithm 8.

Algorithm 8: Computing containment between points and regions

```
computeContain(polygonCellIntersections, cellsAndPoints, partitions) {
    //polygonCellIntersections: see Figure 4.4(b)
    //cellsAndPoints: see Table 4.4
    //partitions: an integer specifying the number of partitions to use

    allGeometries = polygonCellIntersections.union(cellsAndPoints).repartition(partitions, cellid)

    allMatchingPointsAndRegions = allGeometries.mapPartitions((iterator) ⇒ {
        partitionGeoms = iterator.toList
        points = partitionGeoms.filter(geometry ⇒ { geometry.geomtype = "point")
        intersections = partitionGeoms.filter(geometry.geomtype = "polygon")
        matchingPointsForPartition = intersections.flatMap(intersection ⇒ {
            candidatePoints = points.filter(point ⇒ point.cellid = intersection.cellid)
            matchingPointsForRegion = candidatePoints.filter(point ⇒ {
                contains(intersection.coords, point.coords)
            }).map(point ⇒ (point.id, intersection.regionid))
                return matchingPointsForRegion
            })
        return Iterator(matchingPointsForPartition)
    }).map(matchingPair ⇒ (matchingPair.pointid, "TPP,NTPP", matchingPair.regionid)

    allMatchingPointsAndRegions.save() // e.g. (yago:Flanders_Expo, "TPP,NTPP", BEL.2_1)
}
```

The inputs to the program are the points from the YAGO knowledge graph, and the GADM polygon-cell intersections. *cellPolygonIntersections* is actually a subset of the polygon-cell intersections. It isn't necessary to identify all the regions that contain a given point, only the highest level region. For example, it isn't necessary to determine that *Huddersfield is within the United Kingdom*, only that *Huddersfield is within England*, reasoning at query time can determine further transitive containment relations. As a result *cellPolygonIntersections* only features the intersections for the highest level regions. The highest level region isn't always level 1 e.g. the Vatican City doesn't have any sub-divisions. However, identifying the highest level region for each country is trivial, and again for clarity this initial step isn't shown in Algorithm 8

Unlike computing EC relations, a broadcast strategy wasn't used; the volume of data that needed to be broadcast was too great, both the intersection geometries and point geometries needed to be broadcast. The broadcast was possible, but looking-up geometries using such a large broadcast variable slowed computation. Instead, a spatial partitioning approach was adopted. The polygon-cell intersections and points were combined into a single dataset, *allGeomteries*. This combined dataset was re-partitioned using the *cellid*, so that all geometries with the same *cellid* would reside in the same partition.

The *mapPartitions* operation was then used to iterate over each partition. Rather than operating on individual records, *mapPartitions* operates at a partition level, all the records for a specific partition are presented as an iterator, computations are performed and the *mapPartitions* function returns another iterator. Like the broadcast strategy, this approach was adopted to avoid a large join. Although records need to be shuffled when the dataset is re-partitioned by *cellid*, the generation of duplicate records sent to different partitions is avoided.

Within each partition a nested loop is used to iterate over each intersection in an outer loop and over candidate points in an inner loop to identify points that are located in the region's intersection. Results from different partitions are collated and saved.

Analysis There are two stages in Algorithm 8. The first step, re-partitioning, runs in linear time, each object is assigned to a partition. The second step, the *mapPartitions* operation involves filtering the list geometries to obtain a list of points and a list of intersections. These are linear time operations. A nested loop then looks up candidate points for each intersection. This runs in $\mathcal{O}(n \times m)$ time, where n is the number of intersections and m the number of points.

4.4.4 Matching regions with points

In order to take advantage of the richer geometries provided by GADM, it was necessary to match regions in the GADM dataset with equivalent instances in the YAGO dataset. There were several steps involved in matching between the datasets:-

1. Identify point candidates from YAGO using type information e.g. points of the type *schema:Country*.
2. Filter these candidates using their location i.e. only points that are spatially within a GADM region.
3. Test the similarity of the YAGO point's label to the region name from GADM.

For example, if an instance of *schema:Country* has a point representation within the bounds of a GADM country and the point's label has a high text similarity to the GADM country's name, it is assumed to be a match.

The matching is complicated by a number of factors, one of which are the point representations in YAGO. Countries and administrative areas are approximated using a centroid, i.e. a point at the geometric centre of the region. In the case of a multi-polygon region, this centroid can legitimately lie outside the actual bounds of the region's polygons. For example, Malaysia is composed of two parts separated by the South China Sea, the YAGO point representation for Malaysia locates it in the South China Sea. For this reason, minimal bounding boxes were used as approximations of regions as the mbb would cover the centroid in such cases.

Algorithm 9: Matching regions to points

```
1:  matchingRegionsToPoints(pointsLabels, mbbs, types, userType, cleanseArr, threshold) {
2:      //pointsLabels: Table 4.4 joined with the labels for points
3:      //mbbs: minimum bounding boxes for regions in the form (regionid, name, coords, cellid)
4:      //types: The vertically partitioned rdfs:type table
5:      //userType: string specifying the type e.g. schema:AdministrativeArea
6:      //cleanseArr: array of words to cleanse name values e.g. ["principality","governate"...]
7:      //threshold :threshold level for similarity matching

8:      filteredTypes = type.filter(type ⇒ type.object = userType)
9:      filteredPoints = pointsAndLabels.join(filteredTypes, pointsLabels.id=filteredTypes.subject)

10:     //get candidate points
11:     candidates = mbbs
12:         .join(filteredPoints, mbb.cellid = filteredPoints.cellid)
13:         .filter(contains(mbb.coords, point.coords))
14:         .map(candidatePair ⇒ {
15:             score = getSimilarityScore(candidatePair.regionname, candidatePair.pointlabel)
16:             return (candidatePair.regionid, candidatePair.pointid, candidatePair.label, score)
17:         })

18:     //reduce to find candidate with the highest score
19:     keyedCandidates = candidates.map(candidatePair ⇒ (candidatePair.regionid, candidatePair))
20:     matchingPoints = keyedCandidates.reduceByKey((candidate1,candidate2) ⇒ {
21:         if candidate1.score > candidate2.score
22:             return candidate1
23:         else
24:             return candidate2
25:         })
26:         .filter(candidatePair ⇒ candidatePair.score > threshold)
27:         .map(candidatePair ⇒ candidatePair.regionid, candidatePair.pointid)

28:     matchingPoints.save() // e.g. (FRA.6_1, yago:Grand_Est)
29: }
```

The inputs to the program were generated during the initial parsing e.g. the *rdf:type* table or were easily constructed by joining tables generated during the initial parsing e.g. *pointsLabels* was built by joining the YAGO points (Table 4.4) to the vertically partitioned YAGO *rdfs:label* table. The mbbs were also computed at the parsing stage (see Section 4.3.2), and then had a cell covering generated for them. Creating these datasets involve joining comparatively simple data structures and sizes. They use basic operations and don't present performance issues. Therefore, the details of these joins aren't shown in Algorithm 9.

Algorithm 9 follows the three steps outlined above. The *pointsLabels* dataset is filtered by type. These filtered points are then joined to the region mbbs on the basis of *cellid*, and filtered to identify points that actually lie within each region's mbb. *mbrContainsPoint()* is a custom function that simply does a containment test using the mbb coordinates and the point's coordinates. This leaves only those points of the correct type that are located within the region's mbb.

The label for these points is compared to the region's name attribute and a similarity score generated. This is implemented using a second custom function *getSimilarityScore()*. This function first cleanses both the label and the region name by removing the type of administrative division from the name e.g. province, oblast, canton etc. This is to encourage a greater number of matches.

For example, without the cleansing there wouldn't be a match between the GADM region with the name *Chubut* and YAGO point with the name *Chubut Province*. Once the names and labels have been cleansed, the text similarity score was obtained using the Jaro-Winkler distance [71].

There can still be multiple candidates for a particular region, so candidates are reduced by the *regionid* and all but the highest scoring point are discarded. Finally only those candidates with a text similarity score over a specified threshold are deemed matches.

The program was executed twice, once for GADM level 0 regions (countries) using matches to *schema:Country*, and again for level 1 regions, using the *schema:AdministrativeArea* type. For level 0 regions a threshold of 0.9 was used and matches were found for 86.7% of countries. For level 1, to encourage a greater number of matches, a threshold value of 0.75 was used, and 59% of GADM regions were matched.

The difference can be explained by the fact that a much higher proportion of layer 1 regions from GADM don't have an equivalent representation in YAGO e.g. Australian states can't be found in YAGO. Furthermore, many towns and cities are also administrative regions. In such cases YAGO features a single instance with multiple properties. In contrast, a dataset such as GADM makes distinctions between different levels of administrative division. As a result the specific region name from GADM often isn't matched with the more general label from YAGO. In the case where a match wasn't found it was still necessary to integrate GADM regions into the combined knowledge graph. This was done by generating a URI based on the region name and specifying a relevant *rdf:type* property. For example, for the GADM region *AUS.7_1 Queensland*, an id of *yago:Queensland* was generated and a *rdf:type* triple added to the knowledge graph.

```
yago:Queensland rdf:type schema:AdministrativeArea
```

The matching is clearly not without weaknesses. Simply generating an id using the region name abuses the notion of a URI, it is possible to generate duplicate identifiers. Furthermore, not all matches are identified. However, linking entities falls outside the main focus of this thesis, and the level of matching obtained is sufficient to allow for effective querying of the resulting knowledge graph.

After identifying matches, it was necessary to update the knowledge graph. If a point from the YAGO knowledge graph matched a region from GADM, the point's *schema:geo* triple was removed from the knowledge graph, so there was a single geometry for each instance. The polygon table, Table 4.3 was updated to use the new matching YAGO id for each region. Finally, the RCC8 relations computed in Sections 4.4.2 and 4.4.3 were updated to use the YAGO id rather than the GADM region id. Again these are simply a series of simple join operations, the details of which aren't shown.

4.4.5 Generating the final knowledge graph

Generating containment relations between GADM regions Containment relations between regions are already implicit in geospatial datasets such as GADM in the form of region ids that

follow a hierarchical structure e.g. the region id for Grand Est *FRA.6_1* indicates it is part of France. {TPP, NTPP} relations were generated between level 1 regions and their parent country using a simple substring operation on the region id.

Integrating *schema:containedInPlace* Relations As described in section 4.3.1 the YAGO knowledge graph features a number of qualitative spatial triples featuring the *containedInPlace* predicate. These relations can be viewed as being equivalent to {TPP, NTPP} relations. However, using these *containedInPlace* relations has a number of issues. Often, the subjects of *containedInPlace* triples also have quantitative representation through a *schema:geo* property. Including these triples can create unnecessary duplication, and in some cases inconsistencies if the point in region computation from section 4.4.3 contradicts a *containedInPlace* triple.

A second source of inconsistencies is if an instance has multiple *containedInPlace* properties that relate to disjoint spatial entities. This tends to be due to ambiguity in the meaning of the *containedInPlace* predicate. For example, the river Rhine has a *containedInPlace* relation with multiple regions, when in fact it overlaps them. This also occurs for roads, railway lines etc. However, not all spatial instances with multiple *containedInPlace* relations describe overlapping, so it isn't as simple as mapping these triples to PO relations.

In light of these issues, integrating the *containedInPlace* data involved filtering out triples where the subject had a *schema:geo* property, and where a spatial entity had multiple *containedInPlace* properties, using only the first of these. This resulted in 147,071 *containedInPlace* relations which were represented as {TPP, NTTP} relations.

These relations were combined with the region containment relations, along with the relations computed in section 4.4.2 to generate an RCC8 network that could be used to answer spatial queries. The final RCC8 network consisted of 1,095,400 nodes (spatial entities) and 1,103,074 relations.

Together with the original YAGO knowledge graph, and the geometries for spatial entities, this formed the enhanced knowledge graph, part of which was shown in Figure 4.2.

4.5 Evaluation

The algorithms presented in Section 4.4 were evaluated in terms of runtime and scalability. There are no similar systems i.e. distributed approaches to computation of RCC8 relations for large scale datasets, that can be used as a comparison. Also, it is difficult to evaluate the effectiveness of the specific algorithm design choices made in Section 4.4 e.g. using the broadcast strategy or the spatial partitioning implemented in Algorithm 8. Without the use of these approaches, the tasks were unable to complete. Algorithm 9, matching between YAGO and GADM used simple geometries, points and mbbs. The execution didn't require a parallel distributed approach, therefore this algorithm isn't considered in the evaluation.

The algorithms from section 4.4 were implemented using Scala, and tested using computing

clusters provided by Google Dataproc. For the majority of experiments a 4-node cluster was used, with each machine in the cluster having eight virtual CPUs and 52GB of memory. When testing scalability, clusters with a different number of nodes were used, but again each machine had the same 8vCPUs and 52GB of memory.

4.5.1 Datasets

The following datasets, previously described in Section 4.3, were used when running the experiments.

Name	Count	No. of coordinates
GADM level 0 polygons	116,995	32,827,198
GADM level 1 polygon	117,891	37,769,533
YAGO 4 points	957,700	957,700
GADM layer 0 mbbs	256	1024
GADM layer 1 mbbs	3610	14,440

Table 4.5: Input datasets

4.5.2 Runtime

Task	Input	S2 level	Runtime (secs)
Compute cell covering for polygons	GADM layer 0	5	115
Compute cell covering for polygons	GADM layer 1	5	86
Compute cell covering for polygons	GADM layer 0	9	870
Compute cell covering for polygons	GADM layer 1	9	129
Compute cell for points	YAGO points	5	41
Compute cell coverings for mbbs	GADM layer 0	5	53
Compute cell coverings for mbbs	GADM layer 1	5	44
Compute polygon-cell Intersections	GADM layer 0	5	700
Compute polygon-cell Intersections	GADM layer 1	5	497

Table 4.6: Runtime for spatial index generation

Table 4.6 shows the runtime for generating spatial indexes, see Section 4.4.1. Algorithm 5 computed cell coverings for polygons, and polygon-cell intersections. The other tasks shown in Table 4.6 use a similar approach, but process points and mbbs. The S2 levels chosen for the experiments in Table 4.6 are those that were used in subsequent tasks.

Some tasks, computing cells for points and mbbs execute quickly, the geometries of points and

mbbs are so simple that computing cells for these geometries poses few challenges. In fact, a distributed approach isn't really necessary, this computation could be handled by a single machine.

There is a significant difference in runtime between simply identifying the cell covering for polygons and computing the intersection between S2 cells and polygons. This is understandable as computing the intersections is essentially an additional step after the cells have been identified. The initial step of generating a cell covering is handled by the S2 library through its optimised *RegionCoverer* class. The second stage of computing the intersection between these cells and the polygon requires the precise computation of the intersection geometry and is implemented using a more convoluted process of transforming coordinates to library objects, and back again. Furthermore, referring to the analysis in section 4.4.1, in comparison to the first step, this second step involves a much greater number of operations, an intersection computation for each cell a polygon overlaps. There is also a significant difference in runtime between GADM layers, with layer 0, countries, taking longer than layer 1, administrative regions. Again this is to be expected as even though layer 0 is made up of a smaller number of polygons, these polygons have many more vertices. The larger number of simpler polygons in layer 1 can be parallelised more effectively.

Task	Input	Runtime
Compute EC relations between regions	GADM layer 0	145
Compute EC relations between regions	GADM layer 1	165
Compute containment between points and regions	GADM intersections and YAGO points	110

Table 4.7: Runtime for computing RCC8 relations

Table 4.7 shows the runtimes for computing RCC8 relations between spatial entities. The algorithms show a good level of performance. Computing EC relations between all regions in a layer completes in under 3 minutes. Similarly computing containment relations between points and regions takes less than 2 minutes. Considering the overheads of parallel distributed execution, these represent fast times for completing challenging computations. The runtime for computing EC relations for layer 1 is longer than layer 0. Even though the geometries are smaller, a greater number of computations need to be performed for layer 1 regions. For example, although Canada is a larger, more complex geometry, the approximations through using S2 cells means the actual full resolution geometry only needs to be checked once, for an EC relation with USA.

4.5.3 Scalability

The scalability of the tasks in table 4.6 are considered. For these experiments, GADM layer 0 was used as the larger geometries are more challenging to process. Figure 4.5 shows how runtime increases with the S2 level for both generating cell coverings and polygon-cell intersections. This is understandable due to the four fold increase in number of cells that need to be calculated with each increasing S2 level. A maximum S2 level of 6 was used when computing polygon-cell intersections to keep the runtime to a reasonable limit of less than 30 mins.

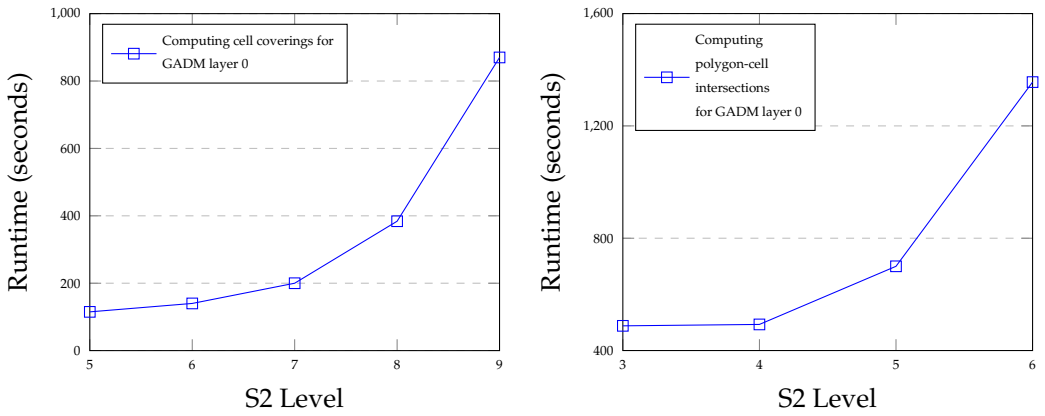


Figure 4.5: Scalability for computing S2 cell coverings and polygon-cell intersections

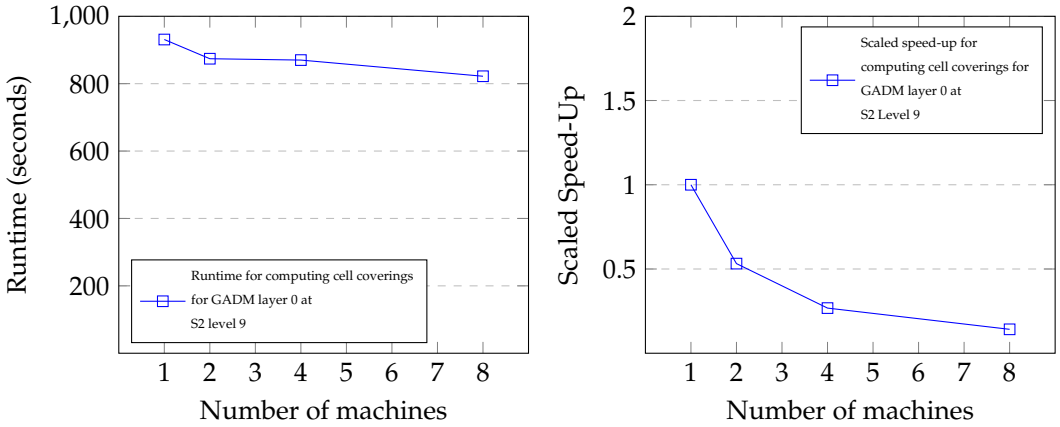


Figure 4.6: Scalability of generating S2 cell coverings with no. of machines

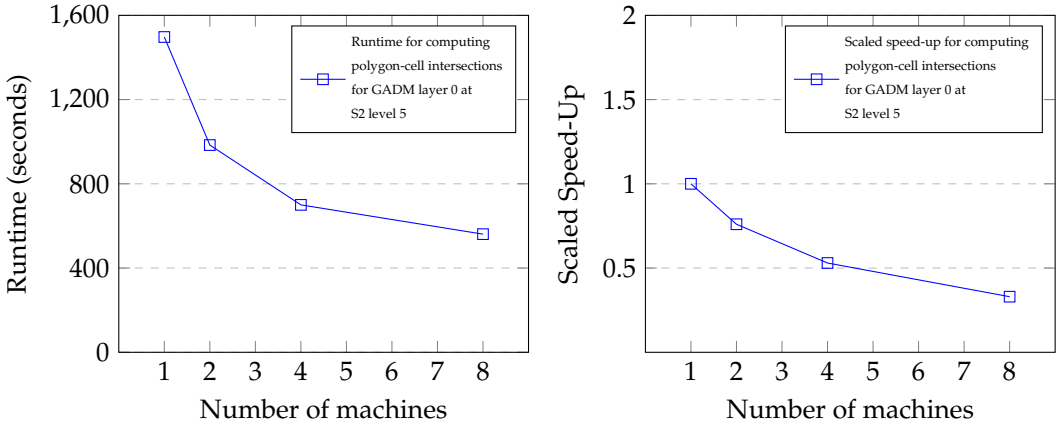


Figure 4.7: Scalability of generating polygon-cell intersections with no. of machines

Figure 4.6 shows scalability for computing cell coverings at S2 level 9 for different sized computing clusters. The results indicate poor scalability, with little improvement in runtime as the cluster size is increased. This is due to the characteristics of the dataset (GADM layer 0) which

doesn't parallelise effectively for this job. Runtime is bound by the longest running cell covering calculation, regardless of how many machines are used in the cluster. Computing the cell coverings for a small number of very large geometries e.g. Canada, Russia, proved to be the limiting factor in improving the runtime of the job. Computing intersections between GADM polygons and S2 cells scales better, see Figure 4.7. Although scaled speed-up, shows less than linear performance, there is a definite decrease in runtime as the size of the cluster is increased. Although there is still the limiting factor of a small number of very large geometries, once the cells for these geometries have been identified, computing the intersection between the cells and polygons can be parallelised leading to faster runtimes when more machines are used.

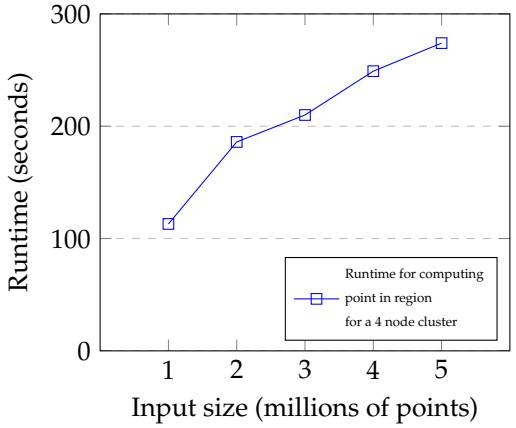


Figure 4.8: Scalability of computing point in region with input size

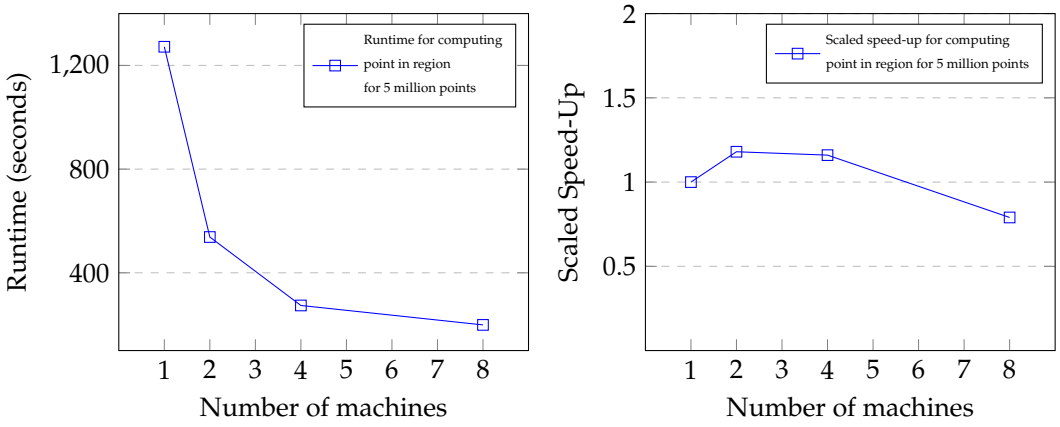


Figure 4.9: Scalability of computing point in region relations with no. of machines

The scalability of computing containment relations between points and regions was tested by using different numbers of points as the input. The English version of the YAGO dataset contains 957,700 points. In order to test the scalability of algorithm 8, this dataset of points was enhanced with additional points from the full version of YAGO. Many of these additional points are of less value when querying as they have fewer predicates and therefore can't support as

wide a range of queries, but they are useful here as they allow us to scale up the dataset for evaluation purposes. Figure 4.8 shows point in region computation for input sizes of between one and five million points. The runtime increases fairly linearly as the input size does, showing the algorithm is able to scale effectively with the input size. This is consistent with the analysis in section 4.4.3. The algorithm has $\mathcal{O}(n \times m)$ time complexity. The experiment involved varying one of the parameters, the number of points (m), so we would expect to see linear increase in time.

Figure 4.9 shows scalability in terms of the number of nodes in the computing cluster. Five million points were used as the input dataset. Initially the algorithm shows super-linear speed before dropping to sub-linear speeds as is often the case when more nodes are added and the benefits of further parallelisation are limited. The initial speed-up provided by two and then four nodes is positive, showing the scalability of the algorithm.

4.6 Related work

Computing RCC8 relations There are no direct comparisons to the algorithms described in Sections 4.4.2 and 4.4.3 i.e. parallel, distributed computation of RCC8 relations. However, there are several established distributed spatial data processing frameworks, and it is worth considering how they compare to the approaches presented here.

Notable distributed spatial data processing systems include SpatialHadoop [21], HadoopGIS [3], SpatialSpark [74], GeoSpark [76] and STARK [26]. These frameworks often have a number of features in common. For example, in order to take advantage of the distributed setting, datasets are partitioned spatially; objects in close proximity are stored in the same partition. Given that spatial queries rarely require the entire dataset, and focus on a particular geographical area, it means that for many queries not all partitions need to be accessed, which can improve performance. Furthermore, for spatial joins, if objects are already co-located in the same partition, less data needs to be shuffled, which again can speed up query execution. Algorithm 8 can be viewed as adopting a similar approach of spatially partitioning a dataset in order to facilitate parallel processing. Many of these frameworks also provide spatial indexing. Typically, indexes are constructed at a local level i.e. within a specific partition, and are implemented using an r-tree. Some experimentation was done with the use of r-trees for the work in this thesis. However, for the programs in this chapter, the simpler grid based indexing approach of cell coverings proved to be more beneficial. Experimental evaluation of frameworks such as STARK suggest impressive performance. For example, using a dataset made up of 322,000 polygons, an evaluation of STARK outperformed existing frameworks, executing point-in-polygon queries in $\approx$ 2 secs. However, such queries can take advantage of not having to access the entire dataset. When the systems were evaluated using more demanding spatial join operations, which are much more analogous to the tasks described in Section 4.4, much smaller datasets, or polygons with smaller number of vertices were used. Presumably this is because of the difficulties in executing join operations using large scale geometries. For example, in [26] experiments were conducted using 38,000 poly-

gons and only 1328 points.

Matching spatial entities The approach to matching spatial entities described in Section 4.4.4 is largely influenced by the LinkedGeoData Project [63]. The project uses OpenStreetMap (OSM) as its source of geographic data. OSM is a freely available map of the world generated via crowd-sourcing e.g. users contributing data from GPS devices, and aerial photography. The LinkedGeo-Data project is concerned with converting this OSM to RDF and interlinking spatial entities from this knowledge graph with entities from other knowledge graphs such as DBpedia and GeoN-ames. The interlinking in the LinkedGeoData project was done using three criteria. Classes from the LinkedGeoData knowledge graph were aligned with classes from other knowledge graphs e.g. cities in LinkedGeoData (lgdo:City) were aligned with instances of DBpedia (dbo:Settlement). Secondly, a text similarity score, between the instance labels in the two datasets was calculated. Finally, the spatial distance between the instances in the different knowledge graphs was taken into account when deciding if there was a match. Clearly the process described in Section 4.4.4 uses similar techniques, but with containment, rather than distance being used to determine matches.

4.7 Conclusions

Summary This chapter has presented algorithms for computing RCC8 relations in a distributed setting and generating an enhanced knowledge graph that can support spatial queries. The effectiveness of these techniques has been shown by implementing the algorithms using GADM, a high resolution vector dataset that covers the entire globe. The scalability has been shown by computing point in region relations for increasingly large numbers of points. A number of key techniques have been presented to facilitate effective parallel processing for these large-scale datasets. Primarily these are the use of broadcast variables and spatial partitioning to mitigate the difficulties of large scale joins.

Future work The techniques presented here could easily be extended to integrate additional layers for the GADM dataset, or to integrate other spatial features. For example, there are vector datasets for rivers and lakes [16]. A similar approach to Algorithm 8 could be used to determine PO or TPP relations between rivers and regions resulting in a richer RCC8 network.

[16] https://www.naturalearthdata.com/downloads/10m-physical-vectors/

Chapter 5

ParQR-QE: A large scale QSTR query engine

5.1 Introduction

Chapter 4 described approaches for generating an enhanced knowledge graph, featuring both quantitative and qualitative spatial information. This chapter considers the querying of such a knowledge graph. A prototype GeoSPARQL query engine, ParQR-QE (Parallel Qualitative Reasoner - Query Engine) has been developed that uses QSTR techniques to provide solutions to spatial queries. ParQR-QE takes an instance based approach to QSTR, and is also able to use hybrid methods (quantitative and qualitative reasoning) when QSTR alone isn't able to answer queries.

The reasoning techniques used in ParQR-QE are described below, along with query execution plans for a number of example queries. The following types of query are considered.

Containment Queries Chapter 4 presented containment query C1, *find all the museums in Belgium*. Another containment query example, query C4 is shown below, *find video game developers that were founded in Canada*.

Query 5.6: Query C4

```
SELECT ?d ?c
  WHERE {
   ?d rdf:type yago:Video_game_developer.
   ?d schema:foundingLocation ?c.
   ?c geo:sfWithin yago:Canada.
}
```

Adjacency Queries Chapter 4 showed containment query A3 *find the French regions that border Belgium*. A similar example is shown in query A4, *find Argentinian Provinces that border Chile*.

Query 5.7: Query A4

```
SELECT ?p
  WHERE {
  ?p rdf:type yago:Provinces_of_Argentina.
  ?p geo:sfTouches yago:Chile.
}
```

Join Queries Query J2 shows a spatial join query. For each US State, the query solution will show Oscar winning actors that were born in the state.

Query 5.8: Query J2

```
SELECT ?a ?s
  WHERE {
  ?a schema:Award yago:Academy_Award_for_Best_Actor.
  ?a schema:birthPlace ?b.
  ?s rdf:type yago:U.S._state.
  ?b geo:sfWithin ?s.
}
```

Window Queries Again, Chapter 4, showed an example of a window query, query W1. Query W2 shows another example which finds the train stations located within a query polygon in South Korea.

Query 5.9: Query W2

```
SELECT ?s
WHERE{
  ?s rdf:type schema:TrainStation.
  FILTER(geof:sfWithin(?s,"POLYGON((129.3693821 35.349243,128.8804905
    35.389557,128.6223118 35.3358005,128.5014621 35.2102298,128.5509006
    35.0395012,128.9299289 34.958505,129.3583957 35.0350036,129.4517795
    35.2281804,129.3693821 35.349243))"^^geo:wktLiteral))
}
```

For clarity, prefixes haven't been shown in the above queries, the full queries can be found in Appendix A.4.

5.2 Instance based reasoning

Using ParQR (Chapter 3) it would be possible to reason using the enhanced knowledge graph presented in Chapter 4. The spatial queries above could then be answered simply by looking up relations for entities specified in the query. This would result in fast query answering, but would

come at the cost of re-computing relations whenever new information is added to the knowledge graph. As we saw in Chapter 3, computing closure for the entire network comes at a time cost - reasoning takes minutes. Instead, the approach used in this Chapter, ParQR-QE, takes an instance based approach that is better able to respond to dynamically changing knowledge graphs.

The instance based approach only derives relations for spatial objects specified in the query. For example, for Query C4 we only need to find relations for *yago:Canada*. For Query J2 we only need to find relations for spatial entities of the type *yago:U.S._state*. In comparison to computing the full closure, limiting the reasoning to expressions from the query limits the size of joins, the number of relations that are inferred, and consequently improves the runtime of the reasoning. However, this alone doesn't improve the performance sufficiently to provide a response time acceptable for query answering. In order to speed-up reasoning further, ParQR-QE implements a broadcast strategy. This is to circumvent the need for the costly shuffle operations which were described in Chapter 3 in both the inference and consistency stages.

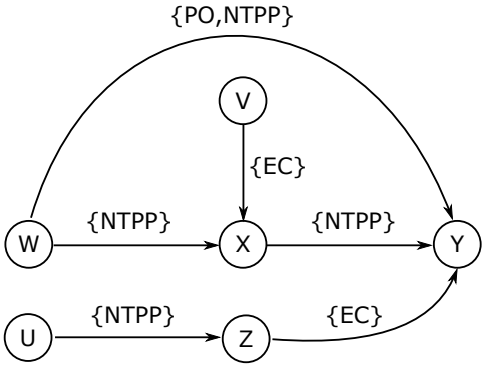


Figure 5.1: Simple RCC8 Network

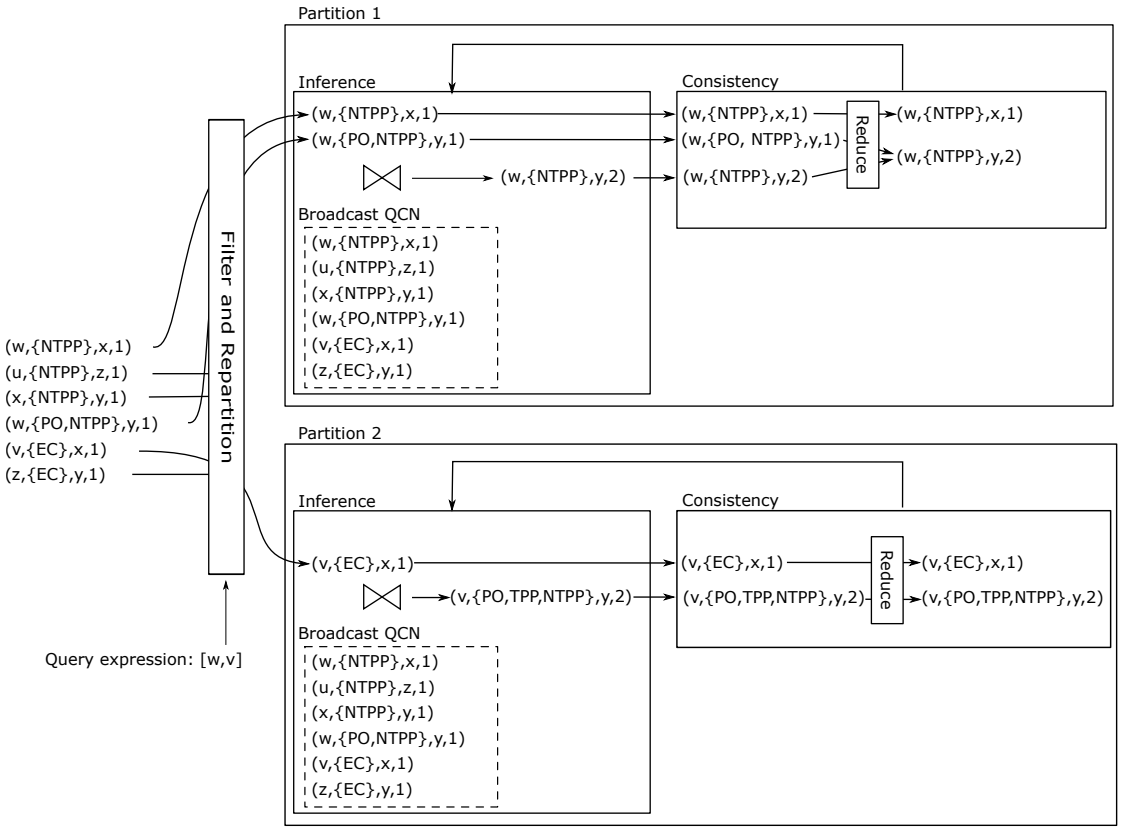


Figure 5.2: Instance based reasoning using ParQR-QE

Figure 5.1 shows a simple RCC8 network, and Figure 5.2 shows an example of instance based reasoning using this QCN. The pre-processing steps (generating look-up tables, generating the reverse of the initial network) aren't shown; they are the same as those presented in Chapter 3. However, at the end of this stage, a copy of the complete input QCN is broadcast to all nodes in the computing cluster.

Query answering is executed by taking an expression from the query and performing instance based reasoning for these entities only. In the example shown in Figure 5.2, these are the variables w and v. The input QCN is filtered to output only those edges with matching head nodes. This subset of edges, the query edges, are re-partitioned so that all the edges with the same head node reside in the same partition. In Figure 5.2, edges featuring w as the head node are sent to partition 1, edges featuring v are sent to partition 2. From this point forward, reasoning takes place within partitions, with no need to shuffle data between different machines or partitions. Where the query only has a single variable e.g. Query C4, reasoning will proceed using a single partition.

At the inference stage, the query edges form the left-hand side of the join. The previously broadcast QCN forms the right hand side of the join. Consistency checking simply involves checking the union of the query edges and newly inferred edges. Figure 5.2 only shows the first iteration of reasoning. However, as with ParQR, the inference and consistency stages would repeat until a fixed point is reached.

Algorithm 10: ParQR-QE: Main program execution

```
query(completeQCNMap, queryVariables, partitions) {
    //completeQCNMap: The complete QCN broadcast to each machine in the cluster
    //queryExpression: An array of spatial entities from the query e.g. [w,v]
    //partitions: An integer specifying the no. of partitions e.g. 64

    queryEdges = completeQCNMap
        .filter((head,edges) ⇒ head ∈ queryExpression)
        .values
        .map(edge ⇒ (edge.headInterval, edge))
        .repartition(new HashPartitioner(partitions))
        .cache()

    result = queryEdges.mapPartitions(partitionEdges ⇒ {
        count = 0
        i=1
        while partitionEdges.count() ≠ count {
            count = partitionEdges.count()
            newEdges = inference(partitionEdges, completeQCNMap, i)
            partitionEdges = consistency(partitionEdges ∪ newEdges, i)
            i++
        }
        return partitionEdges
    }).flatMap(edges ⇒ edges)
     return result
}
```

Algorithm 10 shows the implementation for query answering [1]. The key aspect is the use of the *mapPartitions* operation. The same *while* loop described in Algorithm 2 from Chapter 3 is implemented here but will execute within individual partitions, rather than across all partitions. Figure 5.2 has simplified some aspects which are described in more detail here. For example, the broadcast QCN is structured as a map, with QCN nodes as keys, and an array of edges for this node as the value. This makes it easy to filter the QCN, both when extracting the query triples, and later for the join operation.

[1] Full code listings can be found at https://github.com/mmantle-hud/ParQR-QE

Algorithm 11: ParQR-QE: Inference

```
1:  inference(partitionEdges, completeQCNMap, i) {
2:      //partitionEdges: Query edges for the current partition
3:      //completeQCNMap: The complete QCN keyed by the head node
4:      //i: The iteration number e.g. 1

5:      headEdges = partitionEdges.filter(edge ⇒ edge.distance = i)

6:      partitionJoinedEdges = headEdges.map(headEdge ⇒ {
7:          tailNode = headEdge.tail
8:          tailEdges = completeQCNMap(tailNode)
9:          joinedEdges = tailEdges
10:             .map(tailEdge ⇒ (headEdge,tailEdge))
11:             .filter((headEdge,tailEdge) ⇒ {
12:                 headEdge.head ≠ tailEdge.tail //no joins to self
13:             })
14:         return joinedEdges
15:     }).flatMap(joinedEdges ⇒ joinedEdges)

16:     partitionNewEdges = partitionJoinedEdges.map((headEdge,tailEdge) ⇒ {
17:         inferredRelation = lookUp(headEdge.relation,tailEdge.relation)
18:         distance = headEdge.distance+tailEdge.distance
19:         return (headEdge.head, inferredRelation, tailEdge.tail, distance)
20:     }).filter((newEdge ⇒ newEdge.relation ≠ universalRelation)
21:      .flatMap(newRels ⇒ newRels)

22:     return partitionNewEdges
23: }
```

The inference operation is shown in Algorithm 11. A linear join strategy is used where one side of the join, the broadcast QCN, remains the same with each iteration. This is necessary as if the broadcast QCN were to update, this would need to be re-distributed, which would negate the benefit of this 'reasoning within a partition' approach. Newly inferred relations will always feature a head node that is one of the partition's query variables. There will be no need to re-distribute these edges, consistency checking and further derivations can all be made within the same partition.

The built-in *join* operator in Spark executes across multiple partitions, joining within the partition requires a lower level approach, involving a nested loop, lines 6-15. For each of the partition's query edges, matching edges from the broadcast QCN map are retrieved and joined to the edge. Inference simply involves iterating over these joined edges, and looking up the composition of their relations, see lines 16-21.

Algorithm 12: ParQR-QE: Consistency

```
consistency(partitionEdges, i) {
    //partitionEdges: Query edges for the current partition
    //i: The iteration number e.g. 1

    keyedEdges = partitionEdges
        .map(edge ⇒ (edge.headInterval+'#'+edge.tailInterval, edge))

    keyedEdges.reduceByKey((edgeA,edgeB)⇒
        intersect = edgeA.relation ∩ edgeB.relation

        if |intersect| = 0
            //inconsistency detected
            stop()
        end if

        newDistance = if (edgeA.distance = (i+1) && |edgeB.relation| > |intersect|)
            edgeA.distance
        else if (edgeB.distance = (i+1) and |edgeA.relation| > |intersect|)
            edgeB.distance
        else
            Math.min(edgeA.distance,edgeB.distance)
        endif

        return (edgeA.head, intersect, edgeA.tail, distance)
}
```

Algorithm 12 shows the implementation of the consistency function. This follows the same approach as described in Chapter 3, but again is executed within a partition where it is only necessary to check the consistency of the query edges, not the entire network.

Limitations of the Broadcast Approach Given the performance benefits described in this section for the broadcast approach, it is reasonable to ask why this strategy wasn't used in Chapter 3 when reasoning over complete networks. There are two reasons. First is the usual limitation of the broadcast strategy, the broadcast variable has to fit in memory. For the dataset generated in Chapter 4, this is fine, the reasoner is able to handle large scale QCNs consisting of millions of relations. However, when dealing with even larger QCNs featuring 10s of millions of relations the broadcast variable size becomes a limitation. Secondly, the low level approach to joining edges described above isn't as fast as the Spark optimised sort merge join, when reasoning using a large number of variables or an entire network, the performance benefits of the broadcast approach are limited.

Analysis The algorithms presented in this section are modifications of those presented in Chapter 3. The key difference is that once the initial dataset has been partitioned, and the complete QCN broadcast, execution takes place within partitions, rather than by shuffling data between different machines for operations such as *join* and *reduceByKey*. This includes the main program loop which is also executed within partitions, rather than by the driver. However, the number of iterations within the partitions remains the same as for ParQR i.e. $\mathcal{O}(diam(QCN)$.

The join strategy is different, relying on a broadcast join. Within each partition an outer *map*

operation iterates over the query edges and an inner map operation iterates over the entire QCN to find matching edges that can be joined. For a knowledge graph with n nodes, the worst case is when the QCN is a complete graph with n^2 edges, and the query features all the nodes in the knowledge graph. On a single machine executing using a single partition, this join would run in quadratic time. The outer map iterates over the query edges, n^2. For each query edge, an inner *map* iterates over the broadcast variable of the entire graph i.e. n^2 edges. Of course, in a distributed setting, the query edges are partitioned allowing for parallel execution. Furthermore, a complete graph is an unlikely scenario, as is a query that features all the nodes of the graph as the query expression. Again, as with with ParQR, in the worst case the output from the inference has a total size of $\mathcal{O}(n^3)$ edges. Again these would be distributed between partitions. Consistency involves a simple *map* operation, followed by the *reduceBykey*. Just like with ParQR, the join at the inference stage dominates runtime. This join has a more expensive time complexity than ParQR, but in reality is able to execute faster because it doesn't require a shuffle and one side of the join, the query edges, is almost always smaller.

5.3 Quantitative reasoning for window queries

Using instance based qualitative reasoning alone can be used to answer a wide variety of queries, however, providing solutions to window queries such as query W2 require a hybrid approach, reasoning quantitatively to identify spatial objects within the query window, and then reasoning qualitatively to add in qualitative relations. As discussed in Chapter 4 performing computations using high resolution, large scale geometries is challenging. The following describes the spatial indexing approach used to optimise the quantitative aspect of query answering.

cellid	geomid	id	coords	...
47c4	BEL#1	yago:Belgium	[[[51.36, 3.37...	...
47c4	BEL#2	yago:Belgium	[[[51.42, 4.945...	...
47c4	BEL#3	yago:Belgium	[[[49.49, 5.470...	...
47c4	yago:Waterloo	yago:Waterloo	[[[50.71, 4.38...	...
47c4	yago:Musée_Hergé	yago:Musée_Hergé	[[[50.66, 4.61...	...

cellid	geomid	id	coords	...
47dc	BEL#3	yago:Belgium	[[[51.36, 3.37...	...

cellid	geomid	id	coords	...
47bc	BEL#3	yago:Belgium	[[[51.36, 3.37...	...

cellid	geomid	id	coords	...
47ec	BEL#3	yago:Belgium	[[[51.36, 3.37...	...
47ec	yago:Arlon	yago:Arlon	[[[49.68,5.82...	...

(a) Geometries partitioned by cell id

cellid	geomid	id	...
47c34	yago:Waterloo	yago:Waterloo	...
47c14	yago:Musée_Hergé	yago:Musée_Hergé	...
47eac	yago:Arlon	yago:Arlon	...
47c4c	BEL#1	yago:Belgium	...
47c6c	BEL#2	yago:Belgium	...
47c4c	BEL#3	yago:Belgium	...
47c3c	BEL#3	yago:Belgium	...
47dcc	BEL#3	yago:Belgium	...
...	...	...	...

(b) Cell coverings

Figure 5.3: Spatial indexes for quantitative query answering

Similar to Chapter 4 the query engine uses the S2 Library to identify cells for geometries. Figure 5.3 shows how S2 Cells are used in two ways to enable efficient quantitative query answering. First, using a low level i.e. a small number of large S2 cells, the full geometries of both region polygons and points are partitioned by cell id, and stored by partition to disk in the Parquet format, see Figure 5.3(a). Secondly, cell coverings are generated for both points and region polygons, see Figure 5.3(b).

The justification for this approach is that the full geometries are too large to fit easily into memory without significantly affecting performance. However, spatial queries rarely require the use of all geometric objects in the knowledge graph, only those relevant to the query. By only loading the geometries that are needed, query execution time is much faster. For example, in query W2, we can use the S2Geometry library to identify S2 cells that overlap the query window, and only access the partitions that share a cell id with the query window's cells. Limiting the size of the geometries that are loaded, and subsequent joins involving these geometries, is the greatest single factor affecting query performance for the quantitative reasoning in ParQR-QE.

The cell coverings, Figure 5.3(b), can be used to further reduce the set of geometries that need to be checked. The coverings can also be filtered to identify spatial objects that share cells with the query window. These cell coverings use a higher S2 level and therefore provide a more accurate indication of possible results. Furthermore, in the case of multi-polygon regions, by using the cell

coverings, a number of spatial entities can be discarded if one or more of their polygons are absent from this list of candidates. These cell coverings can then be joined with the full geometries to provide a final list that can be checked for containment within the query polygon. Algorithm 13 shows this implementation.

Different S2 Levels (cell sizes) are used for the geometry partitioning and the cell coverings. Full geometries are stored for each cell the geometry overlaps e.g. in Figure 5.3(a) the geometry for the polygon BEL#3, the main body of Belgium, overlaps four cells, therefore storage for this polygon is duplicated in four different partitions. A balance is needed between using a higher S2 level that results in smaller partition sizes with more duplication, and a smaller S2 level which results in less duplication but larger partitions. In practice, an S2 level of 4 was used for the geometry partitioning. The cell coverings don't store the actual coordinates, so can use a higher resolution that is needed to provide an accurate approximation of spatial objects. Furthermore, the cell coverings fit easily into memory, they are cached as a pre-processing step, allowing for fast access at query time.

Algorithm 13: Quantitative query answering for window queries

```
1:  windowQuery(queryPolygon, cellCoverings, geometries) {
2:      //queryPolygon: The polygons specified in the query e.g. "POLYGON ((4.321423 50.5882119..."
3:      //cellCoverings: See Figure 5.3(b)
4:      //geometries: See Figure 5.3(a)

5:      queryCellCovering = getCellCovering(queryPolygon)
6:      initialCandidates = cellCoverings.filter(cellCovering ⇒ cellCovering.cellid ∈ queryCellCovering)
7:      candidates = filterComplete(initialCandidates)

8:      filteredGeoms = geometries.filter(geometry ⇒ geometry.cellid ∈ queryCellCovering).cache()
9:      initResults = candidates
10:         .join(filteredGeoms, candidate.geomid = filteredGeom.geomid)
11:         .filter(filteredGeom ⇒ queryPolygon.contains(filteredGeom.coords)

12:     quanResults = filterComplete(initResults)

13:     return quanResults
14: }
```

When testing for containment it is necessary to check whether all polygons for a multi-polygon geometry are located within the query window. This check, *filterComplete()*, in Algorithm 13 is performed first using the candidates from the cell coverings (line 7), and a second time using the full geometries (line 12). *filterComplete()* isn't shown in full, but simply involves counting records that have been grouped by id. It is a comparatively fast operation, and is worth doing using the candidate objects to further reduce the size of the subsequent join with the full geometries. Of course, line 12 is necessary to ensure that all parts of a multi-polygon geometry are actually within the query window.

Spark implements lazy loading of datasets. Even though the *geometries* variable has been initialised prior to the execution of the *windowQuery()* function, the geometries table won't be loaded until an action is performed on the table. This is how the query engine can take advantage of the partitioned geometries in Figure 5.3. The first operation performed on the geometries dataset,

line 8, filters by the partition key, only geometries from the specified partition will be loaded. Again, like in Chapter 4 the actual checking of containment (line 11) was performed using the ESRI Geometry API.

5.4 Query execution

Executing GeoSPARQL queries involves processing spatial query triples using ParQR-QE's *query()* function (Algorithm 10) or in the case of window queries, a combination of this function and the *windowQuery()* function (algorithm 13). However, GeoSPARQL queries usually also feature non-spatial query triples. In ParQR-QE non-spatial query triples are mapped to SQL queries which are executed using the vertically partitioned property tables. For example, in Query C4 the non-spatial query triples can be mapped to the following SQL statement.

```
SELECT type.subject AS ?d, foundingLocation.object AS ?c FROM
type JOIN foundingLocation ON type.subject = foundingLocation.subject
```

This mapping is handled with the assistance of Apache Jena [2] which is used to parse the GeoSPARQL queries, and extract query triples and their component elements, so that the mapping to SQL or spatial querying can take place.

It should also be noted that ParQR-QE is very a much a prototype system, designed specifically to test the type of queries shown in the introduction. Many SPARQL and GeoSPARQL features haven't been implemented e.g. restricting on numeric values, ordering, or filtering by alternative spatial properties such as intersect.

The query execution plans for example queries are shown below.

[2] https://jena.apache.org/index.html

5.4.1 Query execution for adjacency queries

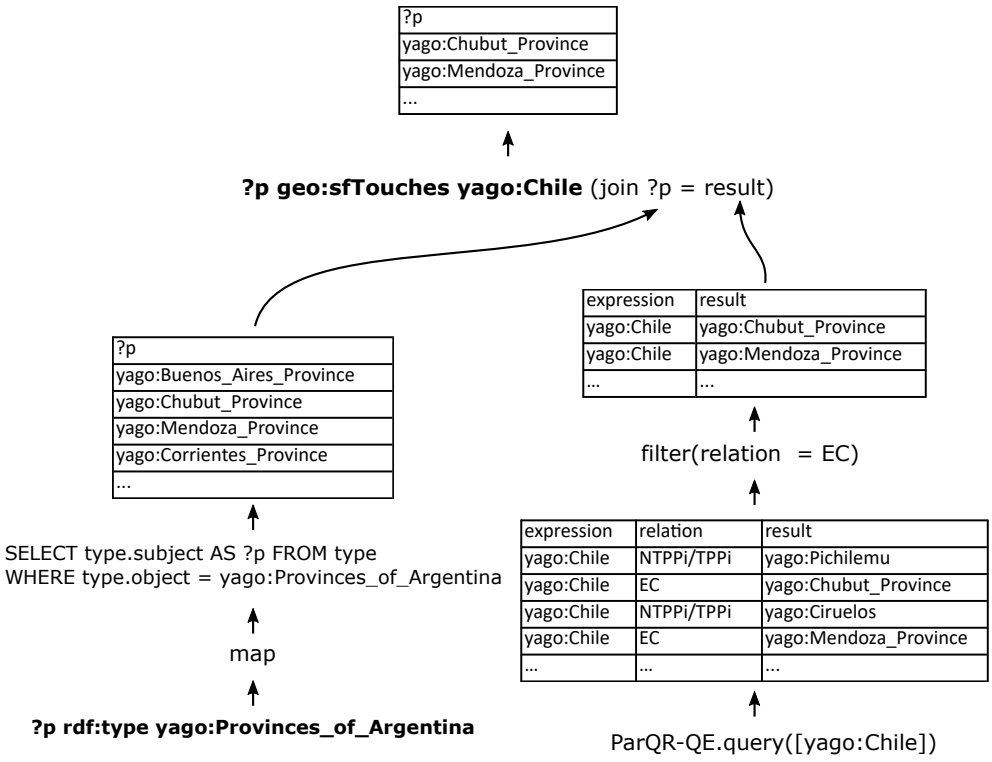


Figure 5.4: Query execution for Query A4

Figure 5.4 shows the execution plan for Query A4. The spatial triple has an expression consisting of a single value, *yago:Chile*. ParQR-QE performs instance based reasoning using this single value. All relations for *yago:Chile* are derived until a fixed point is reached. These relations are then filtered to leave only EC relations. Disjunctive relations e.g. {EC,PO,TPP,NTPP} are filtered out. The results of this qualitative reasoning are then joined with the results from the non-spatial part of the query to provide the query solution. The plan would be the same for containment queries e.g. Query C4 but clearly there would be different query expressions, and filtering would use the {NTPPi,TPPi} relation.

5.4.2 Query execution for spatial join queries

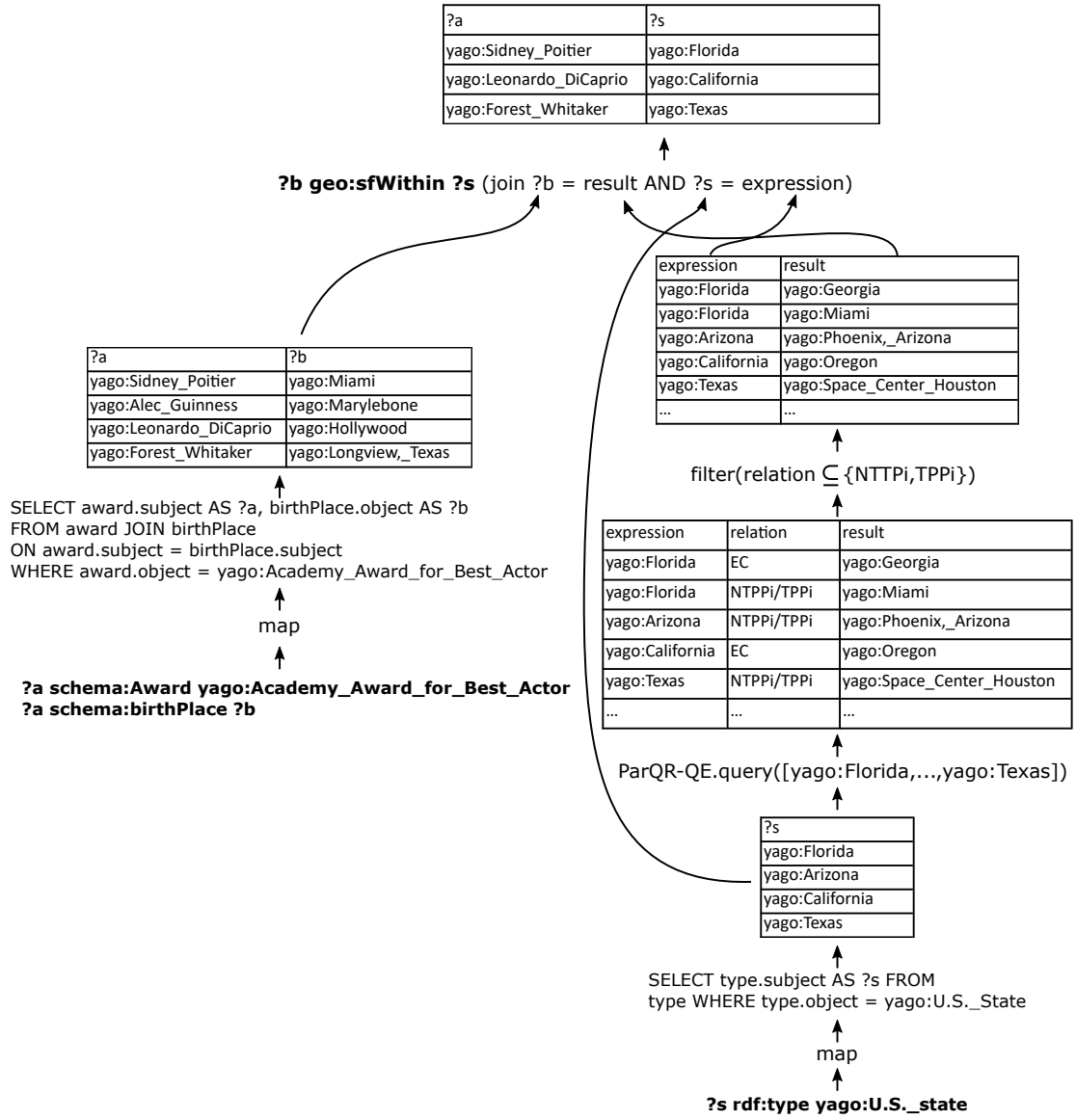


Figure 5.5: Query execution for Query J2

In Query J2, first the non-spatial query triple *?s rdf:type yago:U.S._state* is evaluated. These results are then passed the ParQR-QE for reasoning. Unlike Query A4 reasoning takes place using multiple instances and can therefore take advantage of parallelisation. The reasoning doesn't discriminate between relations, so again filtering needs to be done on the results to identify the containment relations. Two joins are then needed, between the reasoning result and the query variable *?b*, and also between the query expression and the variable *?s*. It may seem as if it isn't necessary to perform this second join as the US states are already present in the results of the qualitative reasoning. However, in the more general case, there may be additional non-spatial results previously joined to this expression that need to be included in the query solution.

5.4.3 Query execution for window queries

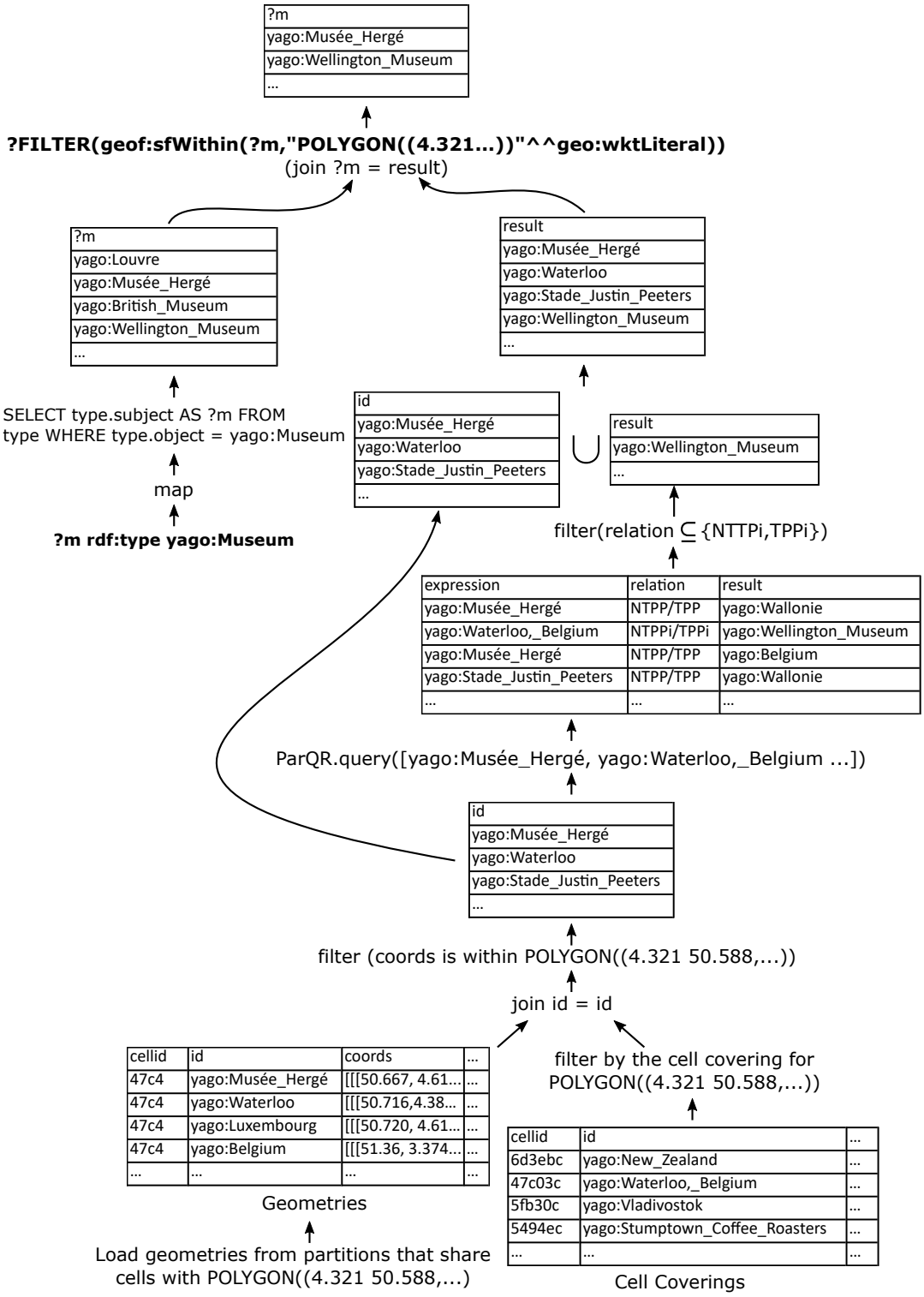


Figure 5.6: Query execution for Query W1

Figure 5.6 considers Query W1. Query execution for window queries first involves quantitative reasoning. As shown in Algorithm 13 the cells of the query window are used to filter the cell coverings table and the geometries. These are joined using the id of the spatial object to restrict

the set of geometries that are actually tested for containment within the polygon. Matching spatial entities are passed to ParQR-QE for qualitative reasoning. The results of the qualitative reasoning are combined with the quantitative results to provide a complete set of spatial results. Finally these are joined with results from the non-spatial query triples *?m rdf:type schema:Museum* to provide the final query results.

5.5 Evaluation

Evaluating ParQR-QE and the algorithms presented in this chapter focuses on two key metrics, query response times and the number of results. In terms of query response times, we are interested in the general performance of qualitative query answering techniques, and also in how response times differ for different types of query. Exploring the number of solution results will give some indication of how successful qualitative approaches are in providing more complete answers in comparison to using purely quantitative data.

The enhanced knowledge graph described in Chapter 4 was used as the basis of the evaluation. This consisted of 1,095,400 spatial entities (nodes in an QCN) and 1,103,074 RCC8 relations. The quantitative data was made up of 234,886 polygons, with a total number of vertices numbering 70,596,731, and 957,700 points.

Twenty example queries were used to evaluate the query engine. These were made up of five containment queries, queries (C1-C5), five adjacency queries (A1-A5), five join queries (J1-J5) and five window queries (W1-W5). The queries can be found in Appendix A.4. For each query, the mean average response time was taken of five successive query executions. These response times don't include pre-processing steps such as broadcasting the input QCN or caching the cell coverings tables. These datasets don't change between query executions, so the broadcast and caching operations can be implemented in advance to speed-up the query response times. A time limit of 5 mins was placed on the execution of queries.

A final experiment was run to test the scalability of the instance based reasoning approach. For this experiment a single query was used, Query J5.

For all experiments except for scalability, the query engine was tested using a cluster of 4 machines. As with previous experiments each machine had eight virtual CPUs and 52GB of memory.

5.5.1 Quantitative query engine

To my knowledge there are no direct comparisons to the ParQR-QE query engine presented in this chapter - a GeoSPARQL query engine capable of querying large scale hybrid knowledge graphs. The related work section in Chapter 4 described large scale spatial querying frameworks e.g. SpatialHadoop and STARK. However, these aren't equipped to handle semantic web data and answer SPARQL queries. The related work section in this chapter also discusses GeoSPARQL implementations. However, these aren't designed to work at scale in a distributed setting.

In order to provide a comparison to ParQR-QE, a quantitative query engine has been developed, this will be referred to as Quan-QE. This system uses the same optimisations described in Section 5.3 i.e. a cell coverings tables loaded into memory and spatially partitioned geometries saved to disk. The query execution plan for Query A4 but executed using Quan-QE is shown in Figure 5.7.

A key difference to the previous query execution plans involving ParQR-QE, is that quantitative query execution can take advantage of early filtering. At the start of the query plan the cell coverings table is joined to the non-spatial query results, to provide the cell coverings for all Argentinian Provinces. This reduced set of cell coverings is then joined to the cell coverings for *yago:Chile* to provide the list of candidate spatial objects i.e. Argentinian Provinces that share cells with Chile. This greatly improves query response times, geometries are not only restricted spatially by using cell coverings, but also by the type of object. This early filtering on object type is an approach widely used in quantitative query answering engines [54].

Once candidate spatial objects have been determined, these candidates are joined with their full geometries, and filtered to only leave those regions that *touch* Chile. This join is the bottleneck in terms of performance. Two joins need to be executed, first to retrieve the coordinates for *yago:Chile* (*geometry.geomid= expression.geomid*), and again to retrieve the coordinates for the candidate provinces (*geometry.geomid = candidate.geomid*). Again, the biggest factor in determining the speed of this operation is the size of the geometries dataset that is loaded from disk. The cell ids of the candidate objects can be used to restrict the partitions that are accessed, limit the size of the geometries dataset, and speed up these joins.

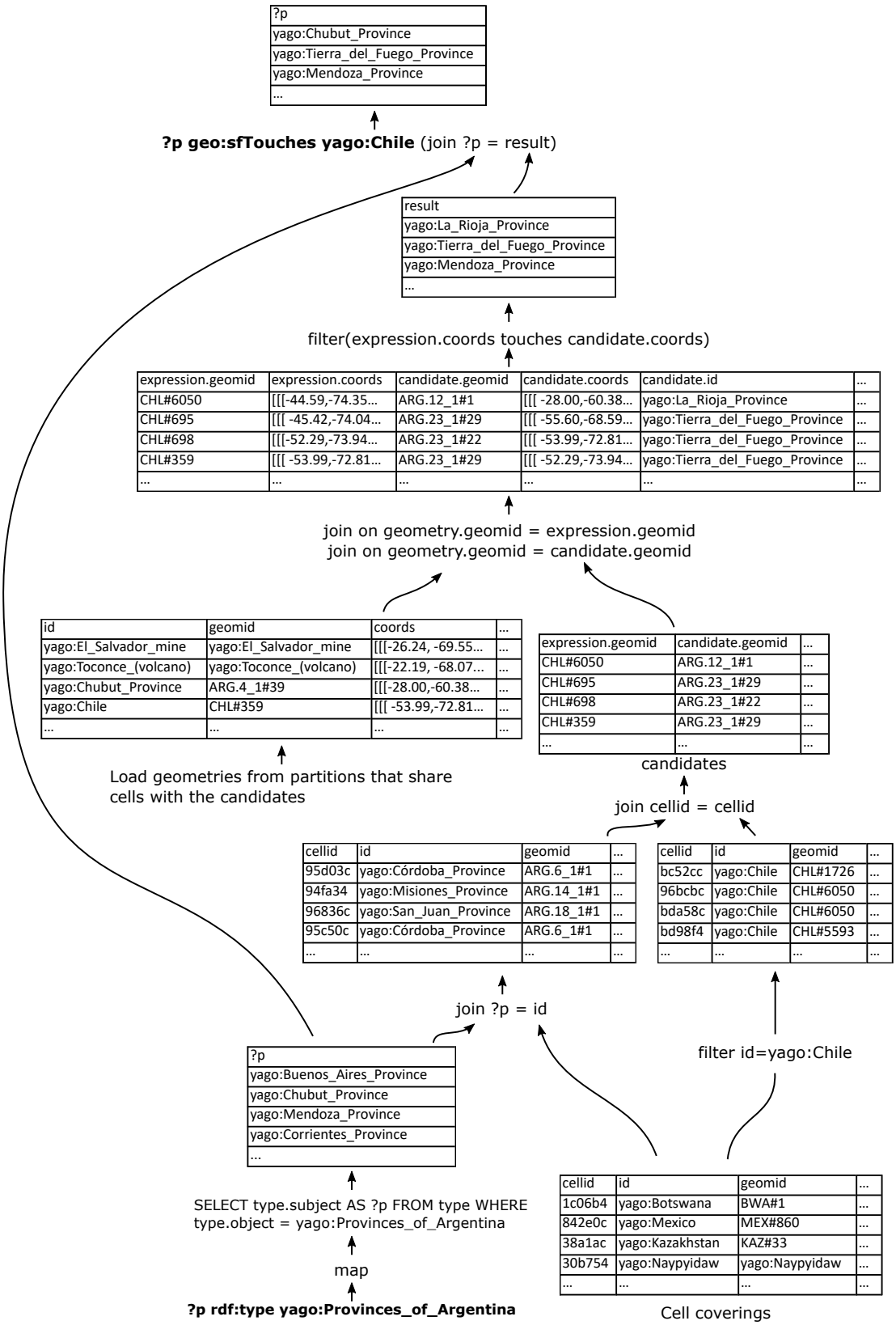


Figure 5.7: Query execution for Query A4 using quantitative reasoning

Containment queries such as Query C4, follow a nearly identical query execution plan. In the case of a spatial join query, the only significant difference is that instead of filtering the cell

coverings table (which is done for *yago:Chile* in Figure 5.7), the cell coverings table is joined with non-spatial query results e.g. for Query J2, the cell coverings would be joined with a table of US States. The rest of the query execution would proceed as in Figure 5.7.

The same queries used to evaluate ParQR-QE were executed using Quan-QE. Again, preprocessing steps e.g. caching the cell coverings table were not part of the measured query execution times.

5.5.2 Containment queries

Table 5.1 show the results for running containment queries.

The query execution times for ParQR-QE are much faster than the reasoning runtimes reported in Chapter 3. The instance based reasoning approach has provided a considerable improvement in reasoning time.

As has been stated previously, irrespective of time complexity, the runtime in distributed parallel applications is dominated by aspects such as serialising/deserialising objects and shuffling data between machines. This is why we see considerably faster exection times for ParQR-QE.

Generally the query response times for ParQR-QE are faster than those for the quantitative query engine. However, often the differences in runtime aren't great, and we should be cautious about drawing definite, more general conclusions about the performance of qualitative vs quantitative reasoning in spatial query answering based on these results. The response times are dependent on the particular knowledge graph being queried, the queries (which often cover wide geographical areas), and specific query engine implementations. It is possible that an alternative approach to the quantitative query answering presented here is able to out perform the qualitative reasoning.

Of more interest is how the query times differ. The query response times for ParQR-QE remain fairly constant. The response time is largely dependent on the number of results generated during reasoning. If reasoning generates lots of inferences, involving several iterations with large joins, the query execution time will be slower. As a general rule reasoning using a small number of instances as a starting point leads to comparatively few derivations, and this is the case for the containment queries. At most 312,874 relations are generated for queries C1-C5 which isn't especially demanding for the reasoner.

In comparison the response times for quantitative query answering vary widely. The dashes for Query C3, *find all the Human Settlements in China*, indicate the query execution exceeded the time limit of 5 mins and was abandoned. The response time for Query C5 *Find actors from the film Casablanca that were born in Bavaria* executes in less than 10 seconds, faster than ParQR-QE. These differences can be explained by differences in the number of full geometries that have to be accessed and checked as part of the query execution. Query C5 focuses on a single German state, a comparatively small geographic area, and subsequently a comparatively small subset of geometries will be loaded. Conversely, Query C3 features China as the query expression, a country that covers a wide area, and requires geometries from many more partitions to be loaded.

Query	ParQR-QE		Quan-QE	
	Time	No. results	Time	No. results
C1	9.102	149	13.779	137
C2	9.025	268	17.880	263
C3	11.015	573	-	-
C4	10.686	9	42.971	9
C5	10.178	1	9.615	1

Table 5.1: Experimental results for containment queries

It can be argued these differences are particular to the approach taken here and other spatial access methods e.g. data driven structures such as R-trees may be more efficient. However, to some extent all quantitative spatial access methods are dependent on characteristics of the spatial objects in the dataset. This is indicative of an area where QSTR spatial answering techniques maybe advantageous. Reasoning using relations isn't dependent on attributes such as the geographic area covered by an object or the complexity of it's geometry.

As expected there are differences in the number of query results, with ParQR-QE finding a greater number of results for queries C1 and C2. These difference arise for two reasons. In some cases data only exists in qualitative form. For example the results for query C2, *find Barcelona footballers that were born in Catalonia*, should include players that were born in the municipality of Navarcles. In the YAGO knowledge graph Navarcles doesn't have a *schema:geo* property, but it does have a *containedInPlace* predicate so is included in the qualitative reasoning. The second reason is inaccuracies in some of the quantitative spatial data. The results should include players born in the coastal town of Mataró. However, the coordinates specified in the *schema:geo* property for Mataró locate it in the Mediterranean Sea, just outside the GADM geometry for Catalonia. However, Mataró also has a *containedInPlace* predicate so footballers born in Mataró are included in the qualitative results.

5.5.3 Adjacency queries

Query	ParQR-QE		Quan-QE	
	Time	No. results	Time	No. results
A1	7.685	6	11.169	6
A2	11.029	4	46.447	4
A3	8.446	2	7.967	2
A4	7.803	10	59.967	10
A5	10.910	15	-	-

Table 5.2: Experimental results for adjacency queries

The ParQR-QE results for adjacency queries in Table 5.2 show greater differences in runtime than for containment queries. This can be explained by the particular queries. For example, Query A1 *find the countries that border Chad* executes in 7.685 secs. Compared to Europe and North America, the YAGO knowledge graph features fewer spatial triples for objects located in Central Africa. Consequently, fewer inferences are made and reasoning executes faster. For Query A1, only 17,259 relations are inferred. Although the differences between queries are greater than for containment queries, the query response times, between 7 and 11 seconds, for ParQR-QE still remain fairly consistent.

Again, the response times for quantitative query answering vary widely, and the explanation is the same as for containment queries. Query A5 *find the countries that border Russia* covers a wide geographic area, requires many partitions to be loaded, and failed to complete. Query A3 *find French regions that border Belgium* executes quickly, as it requires a much smaller set of geometries to be loaded. In comparison, a similar query, query A4 *find Argentinian Provinces that border Chile* has a much longer execution time, not only because Chile covers a wider area than Belgium, but also because of the complexity of Chile's geometry. Chile is multi-polygon made up of over 6000 polygons and more than 2 million vertices.

Unsurprisingly the number of results is the same for qualitative and quantitative query answering. The knowledge graph doesn't feature qualitative adjacency properties, so all results are consequence of computations made using the GADM dataset. However, there are inaccuracies, largely due to the matching discussed in Chapter 4. For example, there are 11 Argentinian Provinces that border Chile, but Neuquén Province wasn't matched correctly, therefore the geometry wasn't linked to the YAGO representation.

5.5.4 Join queries

Query	ParQR-QE		Quan-QE	
	Time	No. results	Time	No. results
J1	10.612	208	11.544	186
J2	31.718	49	16.617	48
J3	9.684	51	11.752	21
J4	154.784	307	-	-
J5	143.524	50	-	-

Table 5.3: Experimental results for join queries

All containment and adjacency queries involve reasoning using a single instance. For spatial join queries the query expression consists of multiple instances, as a result we see wider differences in query execution time. Queries J1 and J3 focus on comparatively sparse parts of the knowledge graph. Query J3 *for each South African Province, find the airports located in the province*, results in the reasoner deriving 74,495 relations. Query J4 *for each nuclear power plant, find the country it is located within* requires instance based reasoning for all 256 countries, results in a reasoner output of 63,516,401 relations, and a considerably longer query response time.

Like containment and adjacency queries quantitative query answering runtime is largely dependent on the size of the geometries that need to be loaded and the size of the join involving these geometries. For example, Query J4 fails to complete within the 5 minute limit. Query J2 (*for each US State, find Oscar winning actors that were born in the state*) is interesting because it shares some similarities with Query A2 (*find US States that border Mexico*) i.e. they focus on a similar geographic area. It might appear surprising that the runtime for the spatial join, J2, is faster than for the adjacency query A2. However, query J2 involves joining points (birthplaces) to polygons (US States). Query A2 involves joining polygons (Mexico) to polygons (US States). The more complex geometries in query A2 result in slower join performance and eventual checking of the geometries. The response times for Query J2 for Quan-QE are faster than for ParQR-QE. This is a consequence of the qualitative reasoner deriving a large number of relations (because USA is a dense part of the KG) making query times slower, and the point based representation of birthplaces in the KG making query times faster for the quantitative implementation.

Again, as with containment queries, differences in the number of results can be attributed the existence of qualitative only information or inaccuracies in spatial object geometries.

5.5.5 Window queries

Query	ParQR-QE (Hybrid reasoning)		Quantitative only	
	Time	No. results	Time	No. results
W1	23.227	4	10.330	3
W2	19.625	53	10.035	53
W3	18.504	2	9.340	2
W4	19.869	2	8.344	0
W5	40.984	50	9.487	41

Table 5.4: Experimental results for window queries

Table 5.4 shows the results for window queries. These were run using the hybrid approach shown in Figure 5.6, and separately using a quantitative only execution. This involved the same query execution plan as in Figure 5.6 but the qualitative reasoning stage was skipped. The query response times for the hybrid approach are much longer than ParQR-QE's response times for containment and adjacency queries. This is for two reasons. First, the query involves executing both qualitative and quantitative reasoning, which are executed successively. Secondly, for the qualitative part, the runtime is largely determined by the number of relations generated during reasoning. Unlike containment/adjacency queries that reason from the basis of a single instance, for window queries, ParQR-QE reasons for all spatial objects that have been found to be quantitatively within the query window e.g. for Query W5, 407 instances are passed to the *query()* function.

In contrast quantitative only query answering runs fast in comparison to other query types because the query polygons generally don't cover as wide an area as the countries and administrative regions used as the expressions in the previous queries. The query polygons used are fairly large e.g for Query W5, the polygon covers 39,305km^2, but these still allow the query engine to limit the number of geometry partitions that are loaded. Furthermore, the types of spatial objects (museums, bridges, train stations etc.) used in these queries are all represented as points, which limits the size of the join, and complexity of the containment tests.

Again qualitative reasoning improves the completeness of the results. Quantitative reasoning identifies objects within the query window. Using these objects as a basis, qualitative reasoning uses $\{NTPPi, TPPi\}$ relations to check for other objects within this window. For queries W1 and W5, the reasoner is able to find additional results using qualitative reasoning.

5.5.6 Scalability

The scalability of the instance based reasoning approach is shown in Figure 5.8. The query with the longest runtime, Query J4, was chosen for this experiment. The containment and adjacency queries would show little speed-up as reasoning is executed within a single partition. It is only in the join and window queries where ParQR-QE can take advantage of parallelisation. The query

was run using computing clusters consisting of 2, 4, 8 and 16 machines, and only executed for ParQR-QE. The results show that although the speed-up isn't optimal, the reasoner does scale effectively, as more machines are added to the cluster, greater parallelisation is possible and query execution times improve significantly. Depending on the dataset and the specific query there will be a point where adding more machines to the computing cluster doesn't significantly improve runtime. For Query J4 we can see the improvement in query response time reduces as we move from 8 to 16 machines. Although the exact results would differ, we would expect to see some speed-up for all the join and window queries. The query expression for all these queries features multiple spatial objects. These can be distributed and the query executed in parallel.

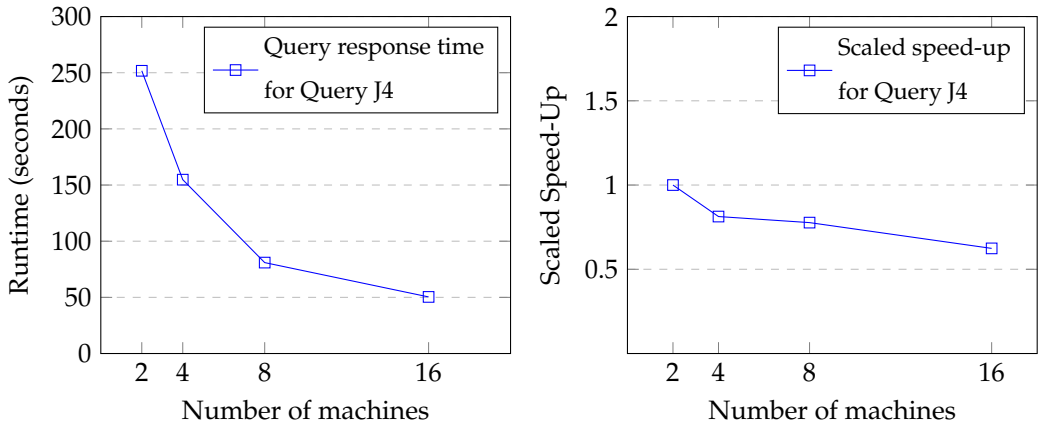


Figure 5.8: Scalability of query response time for Query J4

5.6 Related work

There are no direct comparisons to the system described in this chapter, i.e. a hybrid query engine that integrates QSTR reasoning to answer spatial queries over large scale knowledge graphs. However, there has been work conducted exploring query answering using RCC8 reasoning, hybrid spatial query systems for RDF data, and RDF querying using distributed computing frameworks.

Qualitative spatial reasoning and GIS The idea of a GIS query answering system that uses RCC8 relations was first presented by Bennett in 1996 [11]. He described a prototype system that contained a database of geometrical polygon data and qualitative relations between regions. The system was implemented using Prolog, with queries being entered via the Prolog intepreter e.g. *?- dbq(ec(town,forest))* Queries were answered by adding the query as a constraint to the RCC8 network, and reasoning to determine whether the query is consistent with the database. The prototype system dealt with toy datasets that were very small in scale; it was presented as an example of possible application of RCC8 in GIS.

Stocker and Sirin described PelletSpatial, a qualitative spatial reasoning engine for semantic

web data built on top of the Pellet reasoner [64]. They implemented two different approaches to qualitative spatial reasoning. The first approach was implemented by translating RCC8 relations into OWL-DL class axioms. Queries were then answered by reasoning using the resulting RDF knowledge graph. This approach proved to be impractical even for very small networks. The second approach was similar to the one described in this chapter. The spatial parts of SPARQL queries were answered by reasoning over RCC8 relations e.g. find all regions connected to a specified region, and the non-spatial parts by querying RDF triples that contained additional information about spatial entities e.g. filter these regions based on the size of their population. This reasoner demonstrated the promise of such an approach, reasoning over networks of 10,000 relations, but it also suffered from limitations. Reasoning was limited to the base relations of RCC8, and the reasoner failed to cope with networks of larger sizes.

More recently, Younis et al. presented hybrid geo-spatial query methods using the DBpedia knowledge graph [75]. Faced with the same issues described in Chapter 4 i.e. point based geometries in DBpedia, they used a detailed vector based dataset alongside the spatial data present in DBpedia. For the detailed vector dataset, they limited the area to a single country, the UK, and used vectors from the UK's Ordnance Survey (OS) maps, which were stored in a PostGIS database. Using DBpedia they extracted spatial entities and their point based geometries for specific types of feature e.g. churches, hospitals, and also stored these in a PostGIS database. They were then able to run queries such as *Find Churches within 1km of the River Thames*. To execute such a query, the geometry of the River Thames was looked up using the OS database, this geometry was then used to query the database of DBpedia spatial objects to find matches. Other types of query and further integration with DBpedia was possible e.g. running a query such as *find the mouths of Rivers that cross Oxford*. This involved finding rivers that cross Oxford using the OS database, and then running SPARQL queries against a DBpedia endpoint to find the mouths of these rivers. The system also demonstrated simple hybrid methods. These were used for containment queries e.g. *find all the libraries in Bath*. To execute such a query, the polygon for Bath was extracted from the OS database, the spatial index of DBpedia spatial entities was queried using this geometry. These quantitative results were then combined with the result of running SPARQL queries against the DBpedia endpoint using the containment properties e.g. *?x dbpd:locationCity Bath* . Their experimental results make a strong case for a hybrid approach that integrates more sophisticated polygon based geometries. Using both quantitative and qualitative methods they found that in some cases they were able to nearly double the number of instances retrieved. The system described by Younis et. has a number of significant differences compared to ParQR-QE. They focussed on computation that can be handled by a single machine. Presumably, because of scale and complexity involved in dealing with geometries for the entire globe, they limited their system to the UK only, and a subset of spatial features. Furthermore, even though they took a hybrid approach to query answering, there wasn't any reasoning over qualitative spatial predicates, after obtaining results from quantitative data, they simply ran a separate SPARQL query to integrate entities linked by containment predicates. Finally, their implementation also relies on

simply piping together separate systems, as such they have little fine grained control over issues such as query execution.

GeoSPARQL implementations

An obvious place to look for comparisons with the ParQR-QE system described in this chapter is RDF stores that implement GeoSPARQL. A number of RDF stores support geospatial queries over rdf data e.g Parliament [8], and Strabon [31]. Such RDF stores are capable of executing the queries described in the introduction to this chapter. They typically implement features such as spatial indexing and query optimisation by executing the most selective part of a query first.

RDF triple stores such as Parliament are relevant as examples that can execute GeoSPARQL queries. However, they differ significantly from the work being presented here. These systems all focus on quantitative reasoning. Although the GeoSPARQL standard supports qualitative predicates, this is simply the query syntax. RDF stores don't support qualitative reasoning, they would either need the query to be written to explicitly reference object geometries see Query 5.10 or use query re-writing to transform the Query A4 into Query 5.10.

Query 5.10: Query A4 after query re-writing

```
SELECT ?p
WHERE{
      ?p rdf:type yago:Provinces_of_Argentina.
      ?p geo:hasGeometry ?pgeo
      yago:Chile geo:hasGeometry ?cgeo
      ?pgeo geo:sfTouches ?cgeo.
}
```

Furthermore scalability remains an issue for GeoSpatial RDf stores [27], example workloads have tended to focus on limited geographical extents e.g. a single country or area. As shown in Section 5.5 the parallel, distributed approach as presented here is able to handle large scale spatial data.

Distributed approaches to RDF querying As the size and volume of RDF knowledge graphs has grown, there have been a number of attempts to implement distributed approaches to executing SPARQL queries over large scale knowledge graphs e.g. SHARD [55], H2RDF+ [48], PigSPARQL [56], S2RDF [57]. Spark based system tend to perform better as SPARQL queries often require multiple jobs, and as noted previously, MapReduce is suited to acyclic workflows. The state of the art in terms of distributed RDF querying is S2RDF, which is able to query very large RDF graphs, with a variety of query shapes, in fast times. Like the work presented in this chapter, S2RDF uses vertical partitioning (VP) to store RDF data. S2RDF uses a more sophisticated form of VP than that presented in this chapter; they pre-compute semi-join reductions between pairs of tables to speed up join performance [56]. The performance the S2RDF query engine is impressive. For example, an evaluation using a WatDiv generated dataset, consisting of over 100 million

triples, found that queries were typically executed in less than a second. The obvious limitation of distributed approaches to querying RDF graphs, such as S2RDF, is they don't support spatial queries.

5.7 Conclusions

Summary This chapter presented ParQR-QE, a GeoSPARQL query engine that uses QSTR techniques to answer spatial queries. The query engine uses a novel, instance based reasoning approach to provide fast response times, and is able to integrate quantitative methods for hybrid query answering. The evaluation of the query engine showed that ParQR-QE provides richer query results in comparison to quantitative only methods, and was able to outperform a quantitative approach to spatial query answering. The performance benefits were especially noticeable where quantitative reasoning involved accessing a high volume of complex geometries.

Limitations Although the runtimes for instance based reasoning presented in this chapter are significantly faster than in Chapter 3, a response time 10 secs is considered to be the limit of keeping the user's attention [44] and many of the query response times for ParQR-QE, especially for spatial joins and window queries, are considerably slower

The evaluation used a large scale knowledge graph. However, in future reasoning over even larger knowledge may be desirable. For example, as suggested in Chapter 4, more interesting queries could be answered if the geometries for physical spatial features such as rivers or national parks, were integrated into the knowledge graph. Furthermore, it would be beneficial to integrate further layers of GADM, as users may also be interested in more localised queries e.g. museums within their immediate vicinity rather than all the museums in a country. Using the additional layers that feature smaller administrative regions such as cities would allow for this. Additions like this would significantly increase the size of the knowledge graph, and the response times for the query engine. Furthermore, at some point the broadcast strategy would become unfeasible.

Furthermore, the focus here has been on a very specific scenario involving the YAGO knowledge graph. However, the techniques used by ParQR-QE have widespread applicability, and could feasibly be used for many different temporal or spatial scenarios, which could be larger in scale than the example knowledge graph presented here.

There are other limitations to qualitative spatial query answering presented here. For example, it isn't possible answer distance queries e.g. *find musems within 3km of my current location*.

Further work It may be possible to address some of these limitations. First, there is room to improve the performance of the instance based reasoning:

- In the case of join queries, the reasoner always uses instances from the right-hand side of the query triple e.g. for Query J2, the variable *?s* rather than the variable *?b*. However, if there are fewer edges in the RCC8 network that have a head node from the query expression *?b*,

we are likely to reach a fixed point sooner. Being selective over which query expression to use for reasoning would speed-up query execution.

- Currently, ParQR-QE filters on the type of relation after the reasoning has completed. For transitive relations such as containment, it would be possible to filter during the reasoning, only following lines of reasoning where there is an existing containment relation. This would result in fewer derivations and faster execution times.

Secondly, the performance and potential scalability issues could be addressed simply by precomputing all possible relations. In scenarios involving dynamically changing knowledge graphs, reasoning using the entire knowledge graph (or stable parts of it) could be executed ahead of querying. Further derivations could then be made at query time for recently added information. With many relations already inferred, this would speed-up query response time. The scenario presented in this chapter would be suited to a hybrid approach as the geometries of countries, administrative regions and cities are reasonably stable, and unlikely to change.

Beyond simply improving query response times, a second area of potential future work concerns the relations that are derived. To a certain extent the work presented here uses a carefully orchestrated scenario. The computation of RCC8 relations in Chapter 4 ensured that reasoning would produce meaningful results for query answering. For example, ensuring that at least one containment relation was computed for all the points in the knowledge graph meant queries such as Query C1 would return a complete solution.

In situations where the QCN being queried isn't as comprehensive or we don't have as much control over the construction of the knowledge graph, ParQR-QE may have more utility in presenting possible results, represented as disjunctive relations. This narrowing down of solutions, could then be used as the basis for further decision making or even serve as a form of early filtering for subsequent quantitative computations.

Chapter 6

Conclusions

6.1 Summary

The work presented in Chapter 3 is the first in-depth look at using parallel distributed programming techniques to reason over large scale qualitative constraint networks. This involved the development of a novel, distributed parallel reasoner - ParQR - which has the following notable features:

- Implementation of $\diamond$-consistency algorithms that can take advantage of parallelisation and execute in a distributed environment.
- Optimisations that allow the reasoner to handle large scale knowledge graphs
  - The capacity to use different join strategies dependent on the input knowledge graph.
  - Pre-computation of the results of algebraic operations which speeds up reasoning times.
  - Efficient representation of QCNs which allows the reasoner to handle large scale datasets.

Chapters 4 and 5 considered a specific application of distributed parallel approaches to QSTR - reasoning over large scale knowledge graphs to answer spatial queries.

Chapter 4 considered the problems of spatial data representation in large scale knowledge graphs and used a variety of techniques to generate an enhanced knowledge graph with improved, richer spatial representations. These techniques included:

- Integrating high resolution geometries into existing large scale knowledge graphs.
- Using distributed programming techniques to compute RCC8 relations from large scale, vector based geometries.

There is no published work on the computation of RCC8 relations using parallel distributed programming techniques, as such the work presented in Chapter 4 establishes the first work in this area.

Chapter 5 presented ParQR-QE, an adapted version of ParQR for use in a querying context. The specific use case of query answering meant an instance based approach to reasoning could be used. Using this instance based, broadcast approach, the reasoner was able to execute reasoning in much faster times compared to Chapter 3.

Again the techniques and approaches used were innovative, ParQR-QE is the first of its kind, a large scale QSTR query engine.

6.2 The research question

The research question posed at the start of this thesis was:

> ***Can parallel distributed computing techniques address the challenges of large scale qualitative spatio-temporal reasoning***

The short answer to this question is 'yes'. In Chapter 3 ParQR was evaluated using a variety of large scale knowledge graphs and was able to successfully reason over graphs featuring 150,000,000 relations. ParQR also successfully computed algebraic closure for a number of real world knowledge graphs. As of writing, these are the largest, most challenging real world knowledge graphs in the QSTR research area. Also, importantly, the algorithms presented in Chapter 3 were able to scale effectively, indicating that even larger networks, using larger computing clusters, could be handled. Furthermore, in comparison to existing state of the art reasoners, ParQR was able to reason over much larger networks. Traditional approaches to QSTR struggled with anything other than the smallest networks. Even approaches adopted specifically to address the challenge of large scale QSTR, such as triangulation and graph partitioning, were limited in terms of the size and topology of the knowledge graphs they could successfully reason over.

However, there are a number of caveats for the use of parallel distributed computing techniques to implement QSTR

- The distributed approaches presented in Chapter 3 did struggle with some knowledge graphs. Specifically, there were two factors that exposed the limitations of ParQR
  - Dense graphs where each node has a high degree, results in a vast number of derivations computed in a short number of cycles.
  - Skewed datasets where some nodes in the knowledge graph have a disproportionately large number of relations leads to some partitions becoming very large which creates an imbalance of work in the computing cluster.

To a certain extent these issues can be mitigated e.g. by using different join strategies to spread the workload over a greater number of iterations. However, the possibility of unmanagable knowledge graphs still exists. Despite this, the techniques presented in Chapter 3 are still a significant step forward in comparison to existing approaches to large scale QSTR.

Whereas Chapter 3 considered the general case of consistency for QCNs, Chapters 4 and 5 considered a specific example, and considered whether parallel, distributed techniques can be used to create datasets for use in QSTR and be used to query large scale knowledge graphs.

Again, the simple answer to this question is 'yes'. Computing qualitative relations from quantitative data at scale is necessary if systems are to take advantage of both quantitative and qualitative data. The algorithms for computing $\{EC\}$ relations between regions and $\{TPP, NTPP\}$ relations between points and regions presented in Chapter 4 were able to work with high resolution, complex geometries and scale effectively.

Using the resulting knowledge graph, ParQR-QE was able to provide complete query solutions where quantitative only methods could only provide partial results. Furthermore ParQR-QE was able to adapt the techniques presented in Chapter 3 to implement an instance based approach that significantly sped up reasoning times. In comparison to quantitative reasoning, the use of QSTR in ParQR-QE also showed that it has potential in cases where quantitative reasoning involves accessing large, complex geometries. Reasoning using the lighter weight RCC8 representation resulted in faster response times for many queries.

Again, there are limitations to the work presented in Chapter 5.

- The query times, often over 10 seconds, aren't really viable for real world use.
- The reasoning approach was dependent on the use of a broadcast variable. Although ParQR-QE was successful in reasoning over a large scale knowledge graph of over 1 million relations, the broadcast approach can't scale indefinitely. Larger, more complex knowledge graphs will exceed the memory limits of the reasoner.

Despite these limitations, ParQR-QE still shows the utility of distributed parallel QSTR for query answering, namely providing comprehensive query results, and fast response times for some types of query.

Appendix A

Queries used in Experiments

A.1 Containment queries

Query C1

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?m
  WHERE {
  ?m rdf:type schema:Museum.
  ?m geo:sfWithin yago:Belgium.
}
```

Query C2

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?f
WHERE {
  ?f schema:memberOf yago:FC_Barcelona.
  ?f schema:birthPlace ?b.
  ?b geo:sfWithin yago:Cataluna.
}
```

Query C3

```
PREFIX yago: <http://yago-knowledge.org/resource/>
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?h
  WHERE {
   ?h rdf:type yago:Human_settlement.
   ?h geo:sfWithin yago:China.
}
```

Query C4

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?d ?l
  WHERE {
   ?d rdf:type yago:Video_game_developer.
   ?d schema:foundingLocation ?c.
   ?c geo:sfWithin yago:Canada.
}
```

Query C5

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?a ?b
  WHERE {
   schema:Casablanca_(film) schema:actor ?a.
   ?a schema:birthPlace ?b.
   ?b geo:sfWithin yago:Bayern.
}
```

A.2 Adjacency Queries

Query A1

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?c
  WHERE {
  ?c rdf:type schema:Country.
  ?c geo:sfTouches yago:Chad.
}
```

Query A2

```
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?s
  WHERE {
  ?s rdf:type yago:U.S._state.
  ?s geo:sfTouches yago:Mexico.
}
```

Query A3

```
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?r
  WHERE {
  ?r rdf:type yago:Regions_of_France.
  ?r geo:sfTouches yago:Belgium.
}
```

Query A4

```
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?p
  WHERE {
  ?p rdf:type yago:Provinces_of_Argentina.
  ?p geo:sfTouches yago:Chile.
}
```

Query A5

```
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?c
  WHERE {
  ?c rdf:type yago:Country.
  ?c geo:sfTouches yago:Russia.
}
```

A.3 Join Queries

Query J1

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?m ?p
  WHERE {
  ?p rdf:type yago:Prefectures_of_Japan.
  ?m rdf:type schema:Mountain.
  ?m geo:sfWithin ?p.
}
```

Query J2

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?a ?s
  WHERE {
  ?a schema:Award yago:Academy_Award_for_Best_Actor.
  ?a schema:birthPlace ?b.
  ?s rdf:type yago:U.S._state.
  ?b geo:sfWithin ?s.
}
```

Query J3

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?a ?p
  WHERE {
  ?p rdf:type yago:Provinces_of_South_Africa.
  ?a rdf:type schema:Airport.
  ?a geo:sfWithin ?p.
}
```

Query J4

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?p ?c
  WHERE {
  ?p rdf:type yago:Nuclear_power_plant.
  ?c rdf:type schema:Country.
  ?p geo:sfWithin ?c.
}
```

Query J5

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>

SELECT ?c ?b
  WHERE {
  ?b schema:memberOf yago:Council_of_the_Baltic_Sea_States.
  ?c rdf:type schema:Country.
  ?c geo:sfTouches ?b.
}
```

A.4 Window Queries

Query W1

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?m
WHERE{
  ?m rdf:type schema:Museum.
  FILTER(geof:sfWithin(?m,"POLYGON((4.321423 50.5882119,4.6269803
    50.5869041,4.625607 50.7435868,4.3207364 50.7448903,4.321423
    50.5882119))"^^geo:wktLiteral))
}
```

Query W2

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?s
WHERE{
  ?s rdf:type schema:TrainStation.
  FILTER(geof:sfWithin(?s,"POLYGON((129.3693821 35.349243,128.8804905
    35.389557,128.6223118 35.3358005,128.5014621 35.2102298,128.5509006
    35.0395012,128.9299289 34.958505,129.3583957 35.0350036,129.4517795
    35.2281804,129.3693821 35.349243))"^^geo:wktLiteral))
}
```

Query W3

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?b
WHERE{
  ?b rdf:type schema:Bridge.
  FILTER(geof:sfWithin(?b,"POLYGON((-62.4419835 8.6966246,-62.9253819
    8.4630621,-63.2384923 8.4630621,-63.430753 8.3652495,-63.7493565
```

```
    8.240231,-63.7054112 8.0118347,-63.6614659 7.9030293,-63.2165196
    8.0009555,-62.9638341 8.1749882,-62.727628 8.2674123,-62.3870519
    8.3706842,-62.1783116 8.4956609,-62.4419835 8.6966246))"^^geo:
    wktLiteral))
}
```

Query W4

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?h
WHERE{
  ?h rdf:type schema:Hospital.
  FILTER(geof:sfWithin(?h,"POLYGON((35.9535636 34.5728538,35.6129874
    34.4687574,35.53059 34.2647148,35.5168571 34.0442466,35.6651725
    33.8893489,35.931591 33.8003833,36.2501945 33.8893489,36.4122428
    34.0715522,36.5275993 34.1579619,36.4864005 34.4053307,36.2776603
    34.4778144,36.2199821 34.5796381,35.9700431 34.6180725,35.9535636
    34.5728538))"^^geo:wktLiteral))
}
```

Query W5

```
PREFIX schema: <http://schema.org/>
PREFIX yago: <http://yago-knowledge.org/resource/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geof: <http://www.opengis.net/def/function/geosparql/>

SELECT ?o
WHERE{
  ?o rdf:type schema:EducationalOrganization.
  FILTER(geof:sfWithin(?o,"POLYGON((86.0366359 21.2718418, 85.0918116
    21.0669432, 84.7292628 20.6768591, 84.8940577 19.800688, 85.838882
    19.4488513, 87.0583644 19.873029, 87.1682277 20.9951618, 86.0366359
    21.2718418))"^^geo:wktLiteral))
}
```

Bibliography

[1] Daniel J. Abadi et al. "Scalable Semantic Web Data Management Using Vertical Partitioning". In: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Ed. by Christoph Koch et al. ACM, 2007, pp. 411–422. URL: http://www.vldb.org/conf/2007/papers/research/p411-abadi.pdf.

[2] Foto N. Afrati and Jeffrey D. Ullman. "Transitive closure and recursive Datalog implemented on clusters". In: *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*. 2012, pp. 132–143. URL: http://doi.acm.org/10.1145/2247596.2247613.

[3] Ablimit Aji et al. "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce". In: *Proc. VLDB Endow.* 6.11 (2013), pp. 1009–1020. DOI: 10.14778/2536222.2536227. URL: http://www.vldb.org/pvldb/vol6/p1009-aji.pdf.

[4] James F. Allen. "Maintaining Knowledge about Temporal Intervals". In: *Commun. ACM* 26.11 (1983), pp. 832–843. DOI: 10.1145/182.358434. URL: http://doi.acm.org/10.1145/182.358434.

[5] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 1383–1394. DOI: 10.1145/2723372.2742797. URL: https://doi.org/10.1145/2723372.2742797.

[6] Philippe Balbiani, Jean-François Condotta, and Luis Fariñas del Cerro. "Tractability Results in the Block Algebra". In: *J. Log. Comput.* 12.5 (2002), pp. 885–909. DOI: 10.1093/logcom/12.5.885. URL: https://doi.org/10.1093/logcom/12.5.885.

[7] Roman Barták, Robert A. Morris, and Kristen Brent Venable. *An Introduction to Constraint-Based Temporal Reasoning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2014. DOI: 10.2200/S00557ED1V01Y201312AIM026. URL: https://doi.org/10.2200/S00557ED1V01Y201312AIM026.

[8] Robert Battle and Dave Kolas. "Enabling the geospatial Semantic Web with Parliament and GeoSPARQL". In: *Semantic Web* 3.4 (2012), pp. 355–370. DOI: 10.3233/SW-2012-0065. URL: https://doi.org/10.3233/SW-2012-0065.

[9] Peter van Beek. "Approximation Algorithms for Temporal Reasoning". In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*. Ed. by N. S. Sridharan. Morgan Kaufmann, 1989, pp. 1291–1296. URL: http://ijcai.org/Proceedings/89-2/Papers/071.pdf.

[10] Peter van Beek and Dennis W. Manchak. "The Design and Experimental Analysis of Algorithms for Temporal Reasoning". In: *J. Artif. Intell. Res.* 4 (1996), pp. 1–18. DOI: 10.1613/jair.232. URL: https://doi.org/10.1613/jair.232.

[11] Brandon Bennett, Anthony G. Cohn, and Amar Isli. "A Logical Approach to Incorporating Qualitative Spatial Reasoning into GIS (Extended Abstract)". In: *Spatial Information Theory: A Theoretical Basis for GIS, International Conference COSIT '97, Laurel Highlands, Pennsylvania, USA, October 15-18, 1997, Proceedings*. Ed. by Stephen C. Hirtle and Andrew U. Frank. Vol. 1329. Lecture Notes in Computer Science. Springer, 1997, pp. 503–504. DOI: 10.1007/3-540-63623-4_73. URL: https://doi.org/10.1007/3-540-63623-4%5C_73.

[12] Christian Bliek and Djamila Sam-Haroud. "Path Consistency on Triangulated Constraint Graphs". In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*. Ed. by Thomas Dean. Morgan Kaufmann, 1999, pp. 456–461. URL: http://ijcai.org/Proceedings/99-1/Papers/066.pdf.

[13] Yingyi Bu et al. "HaLoop: Efficient Iterative Data Processing on Large Clusters". In: *Proc. VLDB Endow.* 3.1 (2010), pp. 285–296. DOI: 10.14778/1920841.1920881. URL: http://www.vldb.org/pvldb/vldb2010/pvldb%5C_vol3/R25.pdf.

[14] Aileen Buckley. *Dealing with incomplete data for mapping and spatial analysis*. Accessed: 2021-08-10. URL: https://www.slideshare.net/aileenbuckley/dealing-with-incomplete-data-for-mapping-and-spatial-analysis.

[15] Anthony G. Cohn et al. "Qualitative Spatial Representation and Reasoning with the Region Connection Calculus". In: *GeoInformatica* 1.3 (1997), pp. 275–316. DOI: 10.1023/A:1009712514511. URL: https://doi.org/10.1023/A:1009712514511.

[16] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[17] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[18] Matt Duckham and Lars Kulik. "A Formal Model of Obfuscation and Negotiation for Location Privacy". In: *Pervasive Computing, Third International Conference, PERVASIVE 2005, Munich, Germany, May 8-13, 2005, Proceedings*. Ed. by Hans-Werner Gellersen, Roy Want, and Albrecht Schmidt. Vol. 3468. Lecture Notes in Computer Science. Springer, 2005, pp. 152–170. DOI: 10.1007/11428572_10. URL: https://doi.org/10.1007/11428572%5C_10.

[19] Frank Dylla et al. "Algebraic Properties of Qualitative Spatio-temporal Calculi". In: *Spatial Information Theory - 11th International Conference, COSIT 2013, Scarborough, UK, September 2-6, 2013. Proceedings*. Ed. by Thora Tenbrink et al. Vol. 8116. Lecture Notes in Computer Science. Springer, 2013, pp. 516–536. DOI: 10.1007/978-3-319-01790-7_28. URL: https://doi.org/10.1007/978-3-319-01790-7%5C_28.

[20] Lisa Ehrlinger and Wolfram Wöß. "Towards a Definition of Knowledge Graphs". In: *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems - SEMANTiCS2016 and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16) co-located with the 12th International Conference on Semantic Systems (SEMANTiCS 2016), Leipzig, Germany, September 12-15, 2016*. Ed. by Michael Martin, Martıé Cuquet, and Erwin Folmer. Vol. 1695. CEUR Workshop Proceedings. CEUR-WS.org, 2016. URL: http://ceur-ws.org/Vol-1695/paper4.pdf.

[21] Ahmed Eldawy and Mohamed F. Mokbel. "SpatialHadoop: A MapReduce framework for spatial data". In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. Ed. by Johannes Gehrke et al. IEEE Computer Society, 2015, pp. 1352–1363. DOI: 10.1109/ICDE.2015.7113382. URL: https://doi.org/10.1109/ICDE.2015.7113382.

[22] Berihun Fekade et al. "Probabilistic Recovery of Incomplete Sensed Data in IoT". In: *IEEE Internet Things J.* 5.4 (2018), pp. 2282–2292. DOI: 10.1109/JIOT.2017.2730360. URL: https://doi.org/10.1109/JIOT.2017.2730360.

[23] Andrew U. Frank. "Qualitative Spatial Reasoning: Cardinal Directions as an Example". In: *Int. J. Geogr. Inf. Sci.* 10.3 (1996), pp. 269–290. DOI: 10.1080/02693799608902079. URL: https://doi.org/10.1080/02693799608902079.

[24] Martin Charles Golumbic and Ron Shamir. "Complexity and Algorithms for Reasoning about Time: A Graph-Theoretic Approach". In: *J. ACM* 40.5 (1993), pp. 1108–1133. DOI: 10.1145/174147.169675. URL: https://doi.org/10.1145/174147.169675.

[25] William Gropp et al. "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard". In: *Parallel Comput.* 22.6 (1996), pp. 789–828. DOI: 10.1016/0167-8191(96)00024-5. URL: https://doi.org/10.1016/0167-8191(96)00024-5.

[26] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. "The STARK Framework for Spatio-Temporal Data Analytics on Spark". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*. Ed. by Bernhard Mitschang et al. Vol. P-265. LNI. GI, 2017, pp. 123–142. URL: https://dl.gi.de/20.500.12116/679.

[27] Theofilos Ioannidis et al. "Evaluating Geospatial RDF stores Using the Benchmark Geographica 2". In: *CoRR* abs/1906.01933 (2019). arXiv: 1906.01933. URL: http://arxiv.org/abs/1906.01933.

[28] Holden Karau and Rachel Warren. *High performance Spark: best practices for scaling and optimizing Apache Spark*. " O'Reilly Media, Inc.", 2017.

[29] Jonghoon Kim and Incheol Kim. "Scalable Distributed Temporal Reasoning". In: *Advances in Computer Science and Ubiquitous Computing - CSA/CUTE 2017, Taichung, Taiwan, 18-20 December*. Ed. by James J. Park et al. Vol. 474. Lecture Notes in Electrical Engineering. Springer, 2017, pp. 829–835. DOI: 10.1007/978-981-10-7605-3_132. URL: https://doi.org/10.1007/978-981-10-7605-3%5C_132.

[30] Manolis Koubarakis et al. "Challenges for qualitative spatial reasoning in linked geospatial data". In: *IJCAI 2011 Workshop on Benchmarks and Applications of Spatial Reasoning (BASR-11)*. 2011, pp. 33–38.

[31] Kostis Kyzirakos et al. "The Spatiotemporal RDF Store Strabon". In: *Advances in Spatial and Temporal Databases - 13th International Symposium, SSTD 2013, Munich, Germany, August 21-23, 2013. Proceedings*. Ed. by Mario A. Nascimento et al. Vol. 8098. Lecture Notes in Computer Science. Springer, 2013, pp. 496–500. DOI: 10.1007/978-3-642-40235-7_35. URL: https://doi.org/10.1007/978-3-642-40235-7%5C_35.

[32] Sanjiang Li and Mingsheng Ying. "Region Connection Calculus: Its models and composition table". In: *Artif. Intell.* 145.1-2 (2003), pp. 121–146. DOI: 10.1016/S0004-3702(02)00372-7. URL: https://doi.org/10.1016/S0004-3702(02)00372-7.

[33] Gerard Ligozat. "Reasoning about Cardinal Directions". In: *J. Vis. Lang. Comput.* 9.1 (1998), pp. 23–44. DOI: 10.1006/jvlc.1997.9999. URL: https://doi.org/10.1006/jvlc.1997.9999.

[34] C. Lutz and M. Milicic. "A Tableau Algorithm for DLs with Concrete Domains and GCIs". In: *Journal of Automated Reasoning* 38.1–3 (2007), pp. 227–259.

[35] Alan K. Mackworth. "Consistency in Networks of Relations". In: *Artif. Intell.* 8.1 (1977), pp. 99–118. DOI: 10.1016/0004-3702(77)90007-8. URL: https://doi.org/10.1016/0004-3702(77)90007-8.

[36] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*. Ed. by Friedhelm Meyer auf der Heide and Michael A. Bender. ACM, 2009, p. 48. DOI: 10.1145/1583991.1584010. URL: https://doi.org/10.1145/1583991.1584010.

[37] Matthew Mantle, Sotirios Batsakis, and Grigoris Antoniou. "Large scale distributed spatiotemporal reasoning using real-world knowledge graphs". In: *Knowl. Based Syst.* 163 (2019), pp. 214–226. DOI: 10.1016/j.knosys.2018.08.035. URL: https://doi.org/10.1016/j.knosys.2018.08.035.

[38] Matthew Mantle, Sotirios Batsakis, and Grigoris Antoniou. "Large Scale Reasoning Using Allen's Interval Algebra". In: *Advances in Soft Computing - 15th Mexican International Conference on Artificial Intelligence, MICAI 2016, Cancún, Mexico, October 23-28, 2016, Proceedings, Part II*. Ed. by Obdulia Pichardo-Lagunas and Sabino Miranda-Jiménez. Vol. 10062. Lecture Notes in Computer Science. Springer, 2016, pp. 29–41. DOI: 10.1007/978-3-319-62428-0_3. URL: https://doi.org/10.1007/978-3-319-62428-0%5C_3.

[39] Ugo Montanari. "Networks of constraints: Fundamental properties and applications to picture processing". In: *Inf. Sci.* 7 (1974), pp. 95–132. DOI: 10.1016/0020-0255(74)90008-5. URL: https://doi.org/10.1016/0020-0255(74)90008-5.

[40] Sangha Nam and Incheol Kim. "MRQUSAR: A web-scale distributed spatial reasoner using MapReduce". In: *2017 IEEE International Conference on Big Data and Smart Computing, BigComp 2017, Jeju Island, South Korea, February 13-16, 2017*. 2017, pp. 296–303. URL: https://doi.org/10.1109/BIGCOMP.2017.7881681.

[41] Bernhard Nebel. "Solving Hard Qualitative Temporal Reasoning Problems: Evaluating the Efficiency of Using the ORD-Horn Class". In: *Constraints An Int. J.* 1.3 (1997), pp. 175–190. DOI: 10.1007/BF00137869. URL: https://doi.org/10.1007/BF00137869.

[42] Bernhard Nebel and Hans-Jürgen Bürckert. "Reasoning about Temporal Relations: A Maximal Tractable Subclass of Allen's Interval Algebra". In: *J. ACM* 42.1 (1995), pp. 43–66. DOI: 10.1145/200836.200848. URL: https://doi.org/10.1145/200836.200848.

[43] Thomas Neumann and Gerhard Weikum. "The RDF-3X engine for scalable management of RDF data". In: *VLDB J.* 19.1 (2010), pp. 91–113. DOI: 10.1007/s00778-009-0165-y. URL: https://doi.org/10.1007/s00778-009-0165-y.

[44] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[45] Charalampos Nikolaou and Manolis Koubarakis. "Fast Consistency Checking of Very Large Real-World RCC-8 Constraint Networks Using Graph Partitioning". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, 2014, pp. 2724–2730. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8234.

[46] OGC. *GeoSPARQL - A Geographic Query Language for RDF Data*. https://www.ogc.org/standards/geosparql. Accessed: 2021-08-10.

[47] Christopher Olston et al. "Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. Ed. by Jason Tsong-Li Wang. ACM, 2008, pp. 1099–1110. DOI: 10.1145/1376616.1376726. URL: https://doi.org/10.1145/1376616.1376726.

[48] Nikolaos Papailiou et al. “H2RDF+: High-performance distributed joins over large-scale RDF graphs”. In: *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*. Ed. by Xiaohua Hu et al. IEEE Computer Society, 2013, pp. 255–263. DOI: 10.1109/BigData.2013.6691582. URL: https://doi.org/10.1109/BigData.2013.6691582.

[49] David A. Randell, Zhan Cui, and Anthony G. Cohn. “A Spatial Logic based on Regions and Connection”. In: *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, USA, October 25-29, 1992.* 1992, pp. 165–176.

[50] Jochen Renz and Gérard Ligozat. “Weak Composition for Qualitative Spatial and Temporal Reasoning”. In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*. Ed. by Peter van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, 2005, pp. 534–548. DOI: 10.1007/11564751_40. URL: https://doi.org/10.1007/11564751%5C_40.

[51] Jochen Renz and Bernhard Nebel. “Efficient Methods for Qualitative Spatial Reasoning”. In: *J. Artif. Intell. Res.* 15 (2001), pp. 289–318. DOI: 10.1613/jair.872. URL: https://doi.org/10.1613/jair.872.

[52] Jochen Renz and Bernhard Nebel. “On the Complexity of Qualitative Spatial Reasoning: A Maximal Tractable Fragment of the Region Connection Calculus”. In: *Artif. Intell.* 108.1-2 (1999), pp. 69–123. DOI: 10.1016/S0004-3702(99)00002-8. URL: https://doi.org/10.1016/S0004-3702(99)00002-8.

[53] Jochen Renz and Bernhard Nebel. “Qualitative Spatial Reasoning Using Constraint Calculi”. In: *Handbook of Spatial Logics*. Ed. by Marco Aiello, Ian Pratt-Hartmann, and Johan van Benthem. Springer, 2007, pp. 161–215. DOI: 10.1007/978-1-4020-5587-4_4. URL: https://doi.org/10.1007/978-1-4020-5587-4%5C_4.

[54] Philippe Rigaux, Michel Scholl, and Agnès Voisard. *Spatial databases - with applications to GIS*. Elsevier, 2002. ISBN: 978-1-55860-588-6.

[55] Kurt Rohloff and Richard E. Schantz. “Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store”. In: *DIDC'11, Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing, San Jose, CA, USA, June 8, 2011*. Ed. by Tevfik Kosar. ACM, 2011, pp. 35–44. DOI: 10.1145/1996014.1996021. URL: https://doi.org/10.1145/1996014.1996021.

[56] Alexander Schätzle et al. “PigSPARQL: A SPARQL Query Processing Baseline for Big Data”. In: *Proceedings of the ISWC 2013 Posters & Demonstrations Track, Sydney, Australia, October 23, 2013*. Ed. by Eva Blomqvist and Tudor Groza. Vol. 1035. CEUR Workshop Proceedings. CEUR-WS.org, 2013, pp. 241–244. URL: http://ceur-ws.org/Vol-1035/iswc2013%5C_poster%5C_16.pdf.

[57] Alexander Schätzle et al. “S2RDF: RDF Querying with SPARQL on Spark”. In: *Proc. VLDB Endow.* 9.10 (2016), pp. 804–815. DOI: 10.14778/2977797.2977806. URL: http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf.

[58] James G. Schmolze. “Physics for Robots”. In: *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, USA, August 11-15, 1986. Volume 1: Science.* 1986, pp. 44–50. URL: http://www.aaai.org/Library/AAAI/1986/aaai86-008.php.

[59] Konstantin Shvachko et al. “The Hadoop Distributed File System”. In: *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*. Ed. by Mohammed G. Khatib, Xubin He, and Michael Factor. IEEE Computer Society, 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972. URL: https://doi.org/10.1109/MSST.2010.5496972.

[60] Michael Sioutis. “Triangulation Versus Graph Partitioning for Tackling Large Real World Qualitative Spatial Networks”. In: *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, November 10-12, 2014*. IEEE Computer Society, 2014, pp. 194–201. DOI: 10.1109/ICTAI.2014.37. URL: https://doi.org/10.1109/ICTAI.2014.37.

[61] Michael Sioutis and Jean-François Condotta. “Tackling Large Qualitative Spatial Networks of Scale-Free-Like Structure”. In: *Artificial Intelligence: Methods and Applications - 8th Hellenic Conference on AI, SETN 2014, Ioannina, Greece, May 15-17, 2014. Proceedings*. 2014, pp. 178–191. URL: https://doi.org/10.1007/978-3-319-07064-3_15.

[62] Michael Sioutis and Manolis Koubarakis. “Consistency of Chordal RCC-8 Networks”. In: *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*. IEEE Computer Society, 2012, pp. 436–443. DOI: 10.1109/ICTAI.2012.66. URL: https://doi.org/10.1109/ICTAI.2012.66.

[63] Claus Stadler et al. “LinkedGeoData: A core for a web of spatial open data”. In: *Semantic Web* 3.4 (2012), pp. 333–354. DOI: 10.3233/SW-2011-0052. URL: https://doi.org/10.3233/SW-2011-0052.

[64] Markus Stocker and Evren Sirin. “PelletSpatial: A Hybrid RCC-8 and RDF/OWL Reasoning and Query Engine”. In: *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23-24, 2009*. Ed. by Rinke Hoekstra and Peter F. Patel-Schneider. Vol. 529. CEUR Workshop Proceedings. CEUR-WS.org, 2009. URL: http://ceur-ws.org/Vol-529/owled2009%5C_submission%5C_20.pdf.

[65] Ashish Thusoo et al. “Hive - a petabyte scale data warehouse using Hadoop”. In: *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. Ed. by Feifei Li et al. IEEE Computer Society, 2010, pp. 996–1005. DOI: 10.1109/ICDE.2010.5447738. URL: https://doi.org/10.1109/ICDE.2010.5447738.

[66] Marc B. Vilain and Henry A. Kautz. "Constraint Propagation Algorithms for Temporal Reasoning". In: *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, USA, August 11-15, 1986. Volume 1: Science.* 1986, pp. 377–382. URL: http://www.aaai.org/Library/AAAI/1986/aaai86-063.php.

[67] Thomas Vögele, Christoph Schlieder, and Ubbo Visser. "Intuitive modelling of place name regions for spatial information retrieval". In: *International Conference on Spatial Information Theory*. Springer. 2003, pp. 239–252.

[68] W3C. *SPARQL 1.1 Query Language*. https://www.w3.org/TR/sparql11-query/. Accessed: 2021-08-10.

[69] Jan Oliver Wallgrün. "Exploiting Qualitative Spatial Constraints for Multi-hypothesis Topological Map Learning". In: *Spatial Information Theory, 9th International Conference, COSIT 2009, Aber Wrac'h, France, September 21-25, 2009, Proceedings*. Ed. by Kathleen Stewart Hornsby et al. Vol. 5756. Lecture Notes in Computer Science. Springer, 2009, pp. 141–158. DOI: 10.1007/978-3-642-03832-7_9. URL: https://doi.org/10.1007/978-3-642-03832-7%5C_9.

[70] Matthias Westphal, Stefan Wölfl, and Zeno Gantner. "GQR: A Fast Solver for Binary Qualitative Constraint Networks". In: *Benchmarking of Qualitative Spatial and Temporal Reasoning Systems, Papers from the 2009 AAAI Spring Symposium, Technical Report SS-09-02, Stanford, California, USA, March 23-25, 2009*. AAAI, 2009, pp. 51–52. URL: http://www.aaai.org/Library/Symposia/Spring/2009/ss09-02-011.php.

[71] William E Winkler. "String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage." In: (1990).

[72] Diedrich Wolter and Jan Oliver Wallgrün. "Qualitative spatial reasoning for applications: New challenges and the SparQ toolbox". In: *Geographic Information Systems: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2013, pp. 1639–1664.

[73] Mihalis Yannakakis. "Computing the minimum fill-in is NP-complete". In: *SIAM Journal on Algebraic Discrete Methods* 2.1 (1981), pp. 77–79.

[74] Simin You, Jianting Zhang, and Le Gruenwald. "Large-scale spatial join query processing in Cloud". In: *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 34–41. DOI: 10.1109/ICDEW.2015.7129541. URL: https://doi.org/10.1109/ICDEW.2015.7129541.

[75] Eman M. G. Younis et al. "Hybrid Geo-spatial Query Methods on the Semantic Web with a Spatially-Enhanced Index of DBpedia". In: *Geographic Information Science - 7th International Conference, GIScience 2012, Columbus, OH, USA, September 18-21, 2012. Proceedings*. Ed. by Ningchuan Xiao et al. Vol. 7478. Lecture Notes in Computer Science. Springer, 2012, pp. 340–353. DOI: 10.1007/978-3-642-33024-7_25. URL: https://doi.org/10.1007/978-3-642-33024-7%5C_25.

[76] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. "Spatial data management in apache spark: the GeoSpark perspective and beyond". In: *GeoInformatica* 23.1 (2019), pp. 37–78. DOI: 10.1007/s10707-018-0330-9. URL: https://doi.org/10.1007/s10707-018-0330-9.

[77] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. Ed. by Steven D. Gribble and Dina Katabi. USENIX Association, 2012, pp. 15–28. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.