Large Scale Distributed Spatio-Temporal Reasoning using Real-World Knowledge Graphs

Matthew Mantle[a], Sotirios Batsakis[a,b], Grigoris Antoniou[a]

[a] *University of Huddersfield, Huddersfield, UK*
[b] *Technical University of Crete, Chania, Greece*

Abstract

Most of the existing work in the field of Qualitative Spatial Temporal Reasoning (QSTR) has focussed on comparatively small constraint networks that consist of hundreds or at most thousands of relations. Recently we have seen the emergence of much larger qualitative spatial knowledge graphs that feature hundreds of thousands and millions of relations. Traditional approaches to QSTR are unable to reason over networks of such size.

In this article we describe ParQR, a parallel, distributed implementation of QSTR techniques that addresses the challenge of reasoning over large-scale qualitative spatial and temporal datasets. We have implemented ParQR using the Apache Spark framework, and evaluated our approach using both large scale synthetic datasets and real-world knowledge graphs. We show that our approach scales effectively, is able to handle constraint networks consisting of millions of relations, and outperforms current distributed implementations of QSTR.

Keywords: Qualitative Reasoning, Distributed Computing, Parallel Computing, Knowledge Graphs

---

1. Introduction

Qualitative Spatial Temporal Reasoning is concerned with non-metric representations of space and time. Rather than dealing with information that describes the exact time or date of an event, or the precise location of an object in Cartesian space, QSTR systems represent spatial and temporal knowledge as relations between entities. For example, *Alice read her newspaper* ***before*** *Bob had his breakfast* or *Yorkshire is a region* ***within*** *the UK*. The basis for QSTR is the use of a qualitative calculi; a formalism that provides (1) a vocabulary for describing relations and (2) operations that can be used to reason about relations between objects. Well known examples include Allen's Interval Algebra (IA) [1] for describing relations between time intervals and Region Connection Calculus(RCC) [2] for reasoning about regions in topological space. The core reasoning problem for qualitative calculi is to decide consistency. That is, given a set of temporal or spatial propositions, to decide whether it is possible for them all to be true. Consistency checking is implemented in qualitative spatio-temporal reasoners using methods such as path consistency. Applications of QSTR are numerous. It is especially well suited to cases where comprehensive quantitative data is unavailable [3]. This may be because the data originates in qualitative form e.g. natural language, or simply that precise numeric data is missing or incomplete. In the temporal domain, QSTR has been used successfully in planning and scheduling [4]. Notable applications for qualitative spatial reasoning include reasoning and querying in GIS, and robot navigation [5].

Until recently, work in the area of QSTR has largely focussed on comparatively small knowledge bases where datasets consist of hundreds or at most thousands of relations. However, the big data phenomenon means that a number of large scale datasets now exist that pose challenges for traditional approaches to QSTR. For example, the open linked data initiative has given rise to large scale connected datasets, many of which have a spatial or temporal element. The Ordnance Survey (OS), Great Britain's national mapping agency, have published a number of knowledge graphs in RDF format [1] where relations between regions, towns and cities are described using a qualitative vocabulary (*contains*, *touches* etc.). The following is an example of an RDF triple taken from the OS's Boundary Line knowledge graph:

[1]Ordnance Survey Linked Data Platform - http://data.ordnancesurvey.co.uk/

```
<http://data.ordnancesurvey.co.uk/id/country/wales>
<http://data.ordnancesurvey.co.uk/ontology/spatialrelations/touches>
<http://data.ordnancesurvey.co.uk/id/country/england>
```

There is clearly a role for QSTR in checking the consistency of such datasets and reasoning in order to derive new facts and answer spatial queries. Large scale qualitative spatio-temporal reasoning is the main focus of this paper. In particular we describe and evaluate ParQR (Parallel Qualitative Reasoner), an implementation of QSTR techniques for use in a parallel, distributed environment. We show that ParQR is able to handle large-scale qualitative spatio-temporal datasets consisting of millions of relations. We evaluate ParQR using a number of synthetic and real world knowledge graphs, and compare our approach to alternative, existing methods for dealing with the challenges of large-scale QSTR.

The rest of this paper is organised as follows. Section 2 describes the fundamental principles of QSTR and how reasoning takes place using the path consistency algorithm. Existing tools and approaches for reasoning over qualitative spatial temporal datasets are summarised in Section 3. Section 4 describes in detail the ParQR reasoner, and how path consistency is implemented in a distributed, parallel environment. We have evaluated ParQR using a variety of synthetic and real world datasets and a compared it to a number of existing reasoners; the results of these experiments are presented in Section 5. Finally, Section 6 provides a summary and conclusions.

2. Preliminaries

2.1. Qualitative Spatio-Temporal Reasoning

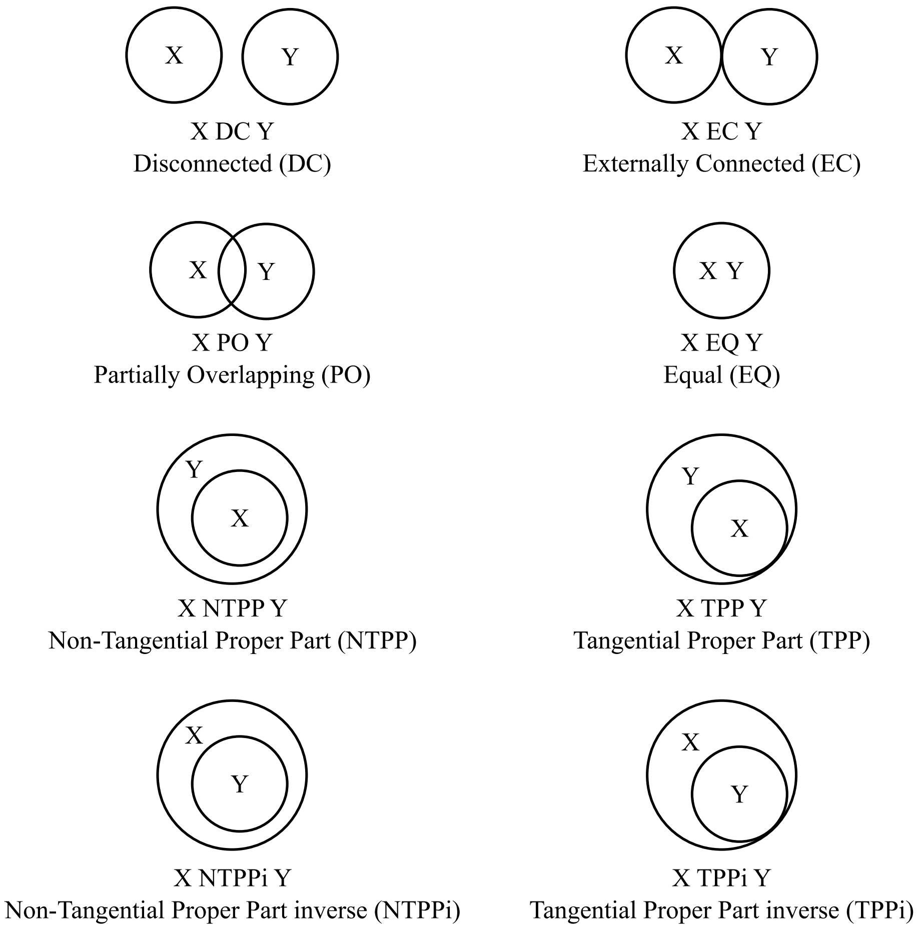
X
Y
X DC Y
Disconnected (DC)
X
Y
X EC Y
Externally Connected (EC)
X
Y
X PO Y
Partially Overlapping (PO)
X Y
X EQ Y
Equal (EQ)
Y
X
X NTPP Y
Non-Tangential Proper Part (NTPP)
Y
X
X TPP Y
Tangential Proper Part (TPP)
X
Y
X NTPPi Y
Non-Tangential Proper Part inverse (NTPPi)
X
Y
X TPPi Y
Tangential Proper Part inverse (TPPi)

Figure 1: RCC8 Atomic Relations

Qualitative constraint calculi provide a formalism for qualitative descriptions of space and time. A calculus consists of a non-empty finite set of atomic relations $\mathcal{B}$ that can exist between entities. For example RCC-8 describes 8 possible relations that can exist between a pair of regions in topological space [2], these are shown in Fig.1. Each atomic relation in a qualitative constraint calculus has an inverse. Taking an example from RCC8, the inverse of *NTPP* is *NTTPi*. Some relations are symmetrical, their inverse is the same relation e.g.*EC*. If the relation between two entities in unknown, a set of relations can be used to describe a disjunction of possible relations that could hold. For example, *X*{*PO,EC*}*Y*, is interpreted as region X *partially overlaps*(PO) or is *externally connected*(EC) to region Y. If no information is known the relation could be any one of the full set of atomic relations, $\mathcal{B}$ e.g. for RCC8 {EC,DC,PO,EQ,TPP,NTPP,TPPi,NTPPi}. The total number of possible relations in a calculus is $2^{\mathcal{B}}$. Other qualitative spatio-temporal calculi follow the same principles, but deal with different domains e.g. Allen's Interval Algebra is concerned with qualitative temporal relations (before, during, after, starts etc.) between time intervals[1].

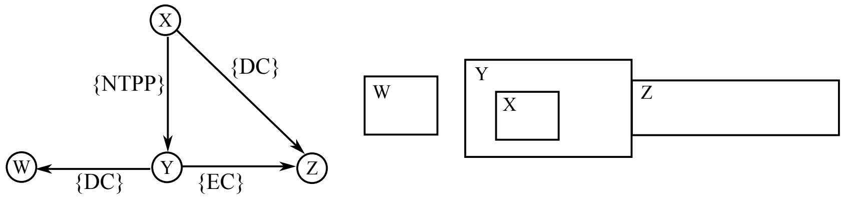

Figure 2: A Simple RCC8 Network and a Consistent Instantiation of the Network

A collection of variables and relations defined between these variables can be represented as a Qualitative Constraint Network, a QCN. Fig.2 shows an RCC8 network. Each node is a variable from the domain, in this case a region in topological space. Labelled edges describe the relation between two variables. When no information is known about the relation between two variables, instead of adding an edge with a label of $\mathcal{B}$, the relation simply isn't shown.

Typically, for a given QCN, we are interested in whether or not the network is *consistent*. That is whether it is possible to assign values to the variables without contravening any of the relations. The network in Fig.2 is consistent, the right of the figure shows a possible (but not unique) instantiation of the network, a possible arrangement of regions that satisfies each of the relations. Note that if the relation between X and Z were *NTTP* instead of *DC*, the network would be inconsistent. It wouldn't be possible to arrange the regions in a way that obeyed all the relations.

Determining consistency for a QCN can be viewed as a type of Constraint Satisfaction Problem (CSP), with the relations being constraints between variables in the network. Consequently, techniques originally formulated to solve CSPs have been used in QSTR to determine the consistency of QCNs. One widely adopted approach is path consistency, specified by the following formula:

$$\forall i, j, k(R_i(i,k) \cap (R_j(i,j) \circ R_k(j,k))) \rightarrow R_s(i,k)$$

Given any three intervals in a constraint network i, j and k, the relation between i and j, and the relation between j and k, imply a relation between i and k. For example, in Fig.2 because X{NTPP}Y and Y{DC}W, X must be disconnected from W i.e. X{DC}W. Deriving a possible relation between two variables in this way is the composition operation($\circ$) for qualitative constraint calculi. Reasoning applications store the results of possible composition operations in a composition table, which is then used to look up inferred relations. Table 1, shows part of the composition table for RCC8 [6]. It is worth noting that the result of composition can be a disjunctive relation e.g. the composition of EC and NTPP is {PO,TPP,NTPP}. Additionally, given two disjunctive relations R_1 and R_2 the composition of these relations is the union of the composition of each atomic relation in R_1 with each atomic relation in R_2.

This inference of relations places constraints on a network as labels on edges are updated to reflect the results of composition. Relations can only be updated if the newly inferred relation is consistent with the existing relation between two nodes. Consistency is checked by taking the intersection of the existing relation and the newly inferred relation. Atomic relations are pairwise disjoint. Therefore, in the case where the intersection of two sets of possible relations is empty, this indicates an inconsistency in the network. For example, if in Fig.2 the network initially contained a relation between X and W of {NTTP}, the intersection of the newly inferred relation X{DC}W with this existing relation would result in $\emptyset$. The network would be inconsistent.

Table 1: Part of the Composition Table for RCC8

$\circ$	DC	EC	NTPP
DC	{*}	{DC,EC,PO,TPP,NTPP}	{DC,EC,PO,TPP,NTPP}
EC	{DC,EC,PO,TPPi,NTPPi}	{DC,EC,PO,TPP,TPPi,EQ}	{PO,TPP,NTPP}
NTPP	{DC}	{DC}	{NTPP}

Reasoning over a qualitative constraint network using the path consistency algorithm is an iterative process. As inferences are made and relations get refined, these refined relations form the basis for new inferences to be made, and further constrain existing relations. The above path consistency operations are applied repeatedly until a fixed point is reached or an inconsistency is detected. Path consistency involves checking 3-node subsets of the network, consequently serial implementations of path consistency run in cubic time in relation to the number of nodes in the constraint network [7].

In the general case, it is important to note that path consistency doesn't guarantee consistency. It is possible for a QCN to be path consistent but not satisfiable. In order to guarantee consistency, we would have to consider consistency between all nodes in a network, *n-consistency*. However, the computational complexity of implementing *n-consistency* is exponential with respect to *n*. An alternative would be to employ depth first, backtracking search alongside path consistency. This is an approach which is adopted by many existing reasoners. However, in the worst-case this also has a runtime which is exponential [8].

2.1.1. Tractable subsets of binary qualitative constraint calculi

Even though path consistency isn't complete, it is still a useful approach that forms the basis for many QSTR applications. This is because there are tractable subsets for many of the most widely used qualitative calculi. A tractable subset is a subset of the 2^B relations which is closed under intersection and composition and for which path consistency is sound and complete. Importantly, for many calculi, there exist tractable subsets that are of practical use. For example the *Ord-Horn* subset of Interval Algebra contains 868 of the 8192 relations present in the full algebra including all 13 atomic relations [8]. Using path consistency in conjunction with the *Ord-Horn* subset can prove sufficient for many real world problems.

2.2. Parallel Distributed Programming

The big data phenomenon is widely acknowledged; the volume of datasets has continued to grow creating challenges for storage, processing and analysis. The sheer size and complexity of these datasets has led to new approaches for the processing of data. Instead of using a single, centralised machine, a parallel, distributed approach is used. A number of different computers are networked together in a cluster. The large dataset is split into smaller chunks, these chunks are distributed to different machines in the cluster, and the task is completed by each machine processing their chuck of data in parallel. The distributed approach allows for the processing of larger datasets and much faster processing than would be possible using a single machine.

Distributed computing brings with it challenges such as how to manage communication between different machines, how to split and distribute data, and how to deal with errors and task failures. Distributed computing frameworks such as Hadoop[2], Spark[3] and Flink[4] have been developed as general purpose infrastructures that handle these common aspects of parallel, distributed computing. Using a framework allows the developer to focus on the specific computation they need to implement instead of concerning themselves with the low level, messy elements of distributed computing.

Built around the MapReduce model, distributed computing frameworks also feature APIs that provide an abstraction for creating data processing programs. The MapReduce model involves two tasks, a *map* task that takes lines from the input dataset and outputs key-value pairs, and a *reduce* task where an aggregate operation is performed on groups of data items with the same key e.g. counting.

[2]https://hadoop.apache.org/
[3]https://spark.apache.org/
[4]https://flink.apache.org/

The focus of this paper, ParQR, has been developed for use with the Spark framework. Spark offers a number of advantages that make it a good fit with QSTR. In additional to the fundamental *map* and *reduce* operations, the Spark API also provides joins, aggregations and filters. The data structure used in Spark, Resilient Distributed Datasets (RDDs), can be cached making Spark especially suitable for iterative applications, such as path consistency, where the same dataset needs to be processed repeatedly [9]. Datasets can be loaded from memory instead of disk, reducing processing times significantly.

3. Related Work

There have been many tools developed for reasoning over Qualitative Constraint Networks. An implementation of path consistency for Interval Algebra was first described by Allen [1] in 1983. Subsequent reasoners introduced a backtracking search element to work alongside path consistency. Ladkin and Reinefeld [10] described a general template for QSTR reasoners, and a number of optimisations for both path consistency and backtracking search have been proposed and evaluated since. These include use of the ORD-Horn class in backtracking [11], skipping techniques, heuristics that determine which order constraints should be processed, pre-computing composition tables [12], and monitoring and acting upon no goods (the reasons for dead-ends in depth first search) [13]. Many of these techniques have been implemented in modern reasoners such as the Generalised Qualitative Reasoner GQR [14], which considered to be state of the art in terms of qualitative spatial temporal reasoning. A limitation of archetypal approaches to solving QCNs concerns memory requirements. Solvers typically use an adjacency matrix representation for a QCN where each element in the matrix represents a relation between connected nodes [14]. Maintaining a complete network in memory results in $O(\mathrm{n}^2)$ memory requirements (where n is the number of nodes). Consequently, large networks consisting of hundreds of thousands or millions of relations expose the memory limits of such tools [15]. Furthermore, even if memory requirements weren't an issue, the search space requirements in path consistency and backtracking search for such large networks make runtime infeasible.

In recent years there has been significant interest in reasoning over large scale QCNs [16],[17] [18] this has led to the implementation of techniques that mitigate the difficulties of reasoning over larger networks.

Bliek and Sam Haround [19] consider the path consistency of triangulated, chordally complete constraint graphs. For triangulated QCNs, checking consistency for a subset of constraints implies consistency for the complete network. There are polynomial time algorithms that are capable of checking chordality and chordally completing a graph by adding *fill edges* so that longer cycles are no longer present. Path consistency is then reduced to simply checking the consistency of triangles in the constraint network, with no need to consider constraints between all nodes. In addition to the chordal requirements, it is important to note that such an approach is only sound and complete for certain calculi. For example, it has been shown that for QCNs with relations limited to one of the maximal tractable subsets of RCC8, the consistency of the chordal graph is equivalent to consistency of the complete graph [20]. Tools have been developed that are able to take advantage of chordal completion in order to reason over larger scale QCNs. For example, Sarissa [21] uses a hash table based adjacency list, that only stores and reasons over the underlying chordal graph for a QCN. Experiments performed on real world RCC8 networks showed the reasoner was able to reason over real world GeoSpatial networks featuring millions of relations. The chordal completion approach is especially suited to relatively sparse networks; in more complex networks triangulation can result in an underlying chordal graph that is so dense that reasoning using this approach offers few efficiency gains. It is also worth noting that, although efficient algorithms exist for detecting chordal completion and adding fill edges, the problem of minimum chordal completion (chordal completion with as few fill edges as possible) is known to be NP-Complete [22].

The other main strategy for dealing with large-scale networks that has generated significant interest from researchers is graph decomposition. For some qualitative calculi, the union of two path consistent networks that agree on common constraints between shared variables results in a larger network such that applying path consistency to the larger network doesn't change any of the constraints [23]. This is known as the *patchwork* property [24] (also called *Atomic Network Amalgamation Property* (aNap) [23]). The patchwork property can be exploited to improve the performance of qualitative reasoners. A large unmanageable QCN can be split into sub-networks. The consistency of individual sub-networks can be checked and conclusions drawn about the consistency of the complete network. Decomposing into sub-networks also means that in total fewer constraints need to be checked as universal relations between variables in different sub-networks are disregarded when deciding satisfiability. This approach is sound and complete for many commonly used tractable subsets such as the ORD-Horn fragment of Interval Algebra and any of the maximal subsets of RCC8 [25]. Li et al. [23] describe a method of recursively decomposing a network with the aim of removing unnecessary relations. The resulting sub-networks were encoded as Boolean formulas and consistency determined using a SAT solver. Experiments comparing their implementation to existing approaches show that using graph decomposition with SAT encoding performed favourably when solving hard instances of networks up to a size of 200 nodes. Sioutis et al. [15] presented an algorithm that decomposed a QCN into biconnected components. The consistency of the sub-components could then be checked by any suitable QCN solver e.g.GQR. Experiments considered real world spatial networks, the largest of which featured hundreds of thousands of relations, and results showed decomposition significantly improved the performance of reasoners.

The approach presented in this paper, ParQR, takes a different approach to the consistency problem for large scale networks. Rather than using triangulation or graph decomposition, a distributed approach is used to spread reasoning tasks over multiple machines thus alleviating some of the problems associated with memory requirements and search space. ParQR isn't the first attempt to develop a parallel, distributed qualitative spatio-temporal reasoner. Mantle et al. [26] implemented path consistency algorithms for Interval Algebra using a distributed approach. Experimental results show they were able to reason over datasets of 10 million relations. MRQUSAR [27] and MRQUTER [28] are distributed spatial/temporal reasoners that use the MapReduce framework. Input sizes of 6 million relations were used in the evaluation presented in [27]. Each of these implementations show limitations. In all cases evaluations only considered synthetic datasets with specific characteristics. In [26] input networks had an average degree of 2, and although at points dense networks were generated these were much smaller in size (thousands of relations) and derived from a sequence of relations which made it much easier for the reasoner to balance the workload. The networks used to test MRQUSAR were also synthetically generated with every node having a degree of exactly one. In fact a spanning tree was generated with no branches. So although both these reasoners show potential, they are limited in terms of scalability, and have only demonstrated the ability to handle sparse networks. There isn't the evidence to suggest they can handle large scale real world networks.

A specific use case discussed in this paper is that of linked open data from the semantic web. Previously, there has been research into the implementation of QSTR techniques using semantic web technologies. Batsakis et al. [29] describe the use of SWRL rules to encode composition tables for both IA and RCC8. Reasoning is then handled by the off the shelf OWL reasoners HermiT [5] and Pellet [6]. Experimental results show their approach is able to reason over tractable RCC8 networks consisting of 100,000 regions in less than 250 seconds. Although this performance is adequate for many applications this is still some way from handling large-scale datasets consisting of millions of relations.

[5]http://www.hermit-reasoner.com/

[6]https://www.w3.org/2001/sw/wiki/Pellet

4. ParQR Implementation

The ParQR system is a distributed implementation of path consistency for qualitative constraint calculi. ParQR is written using the Scala programming language and the Apache Spark framework. It builds on the implementation presented in [26] but provides improvements in a number of key areas [7]. First, the reasoner is generalised. ParQR is able to reason using various qualitative constraint calculi; previously the reasoner only worked with Allen's Interval Algebra. Second, a number of optimisations allow it to handle larger, more complex networks, these include pre-computation of look-up tales for composition and intersection operations, and the use of numeric ids to represent relations resulting in reduced memory requirements. Fig.3 provides a high level overview of the stages involved in ParQR's reasoning. Algorithm 1 describes the primary function used to execute these stages.

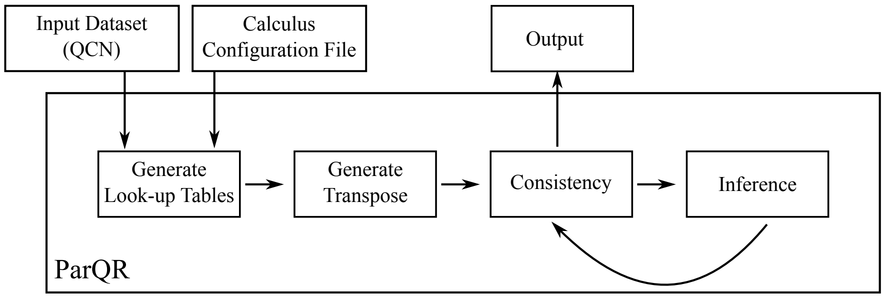

Figure 3: ParQR Stages

The reasoner requires two inputs. An input dataset in the form of a qualitative constraint network and a calculus configuration file. Each line from the input dataset represents a labelled edge from the network in the form of a tuple *(i,R_{ij},j,k)*, where i and j are nodes in the network, R_{ij} the relation between these nodes, and k the distance between the two nodes. All input lines have a value of 1 for k. This distance value is required to limit duplicate inferences

[7]The source code is available at https://github.com/mmantle-hud/ParQR

and is explained in more detail in Section 4.6. The configuration file specifies the atomic relations, the inverse relations, and the basic composition table for a calculus e.g. for Interval Algebra or RCC8. In the first stage of ParQR, the inputs are used to generate complete composition and intersection look-up tables for the network. The transpose of the network is then added so that all possible inferences can be made. The key part of the reasoner is where new relations are derived and the consistency of the network is tested. These two stages, inference and consistency, occur iteratively until a fixed point is reached or an inconsistency in the network is detected. Further details on the stages presented in Fig.3 are provided in the following sections.

Algorithm 1 ParQR

```
ParQR(QCnetwork, calculus)
    //Generate Look-up Tables
    generateLookupTbl(QCnetwork, calculus)
    //Generate Transpose
    QCnetwork=QCnetwork ∪ QCnetwork$^T$
    //Consistency
    QCnetwork=consistency(QCnetwork)
    count=0
    i=1
    while QCnetwork.count() ≠ count
        count=QCnetwork.count()
        //Inference
        newEdges=inference(QCnetwork,i)
        //Consistency
        QCnetwork=consistency(QCnetwork ∪ newEdges)
        i++
    end while
```

4.1. The Inference Stage

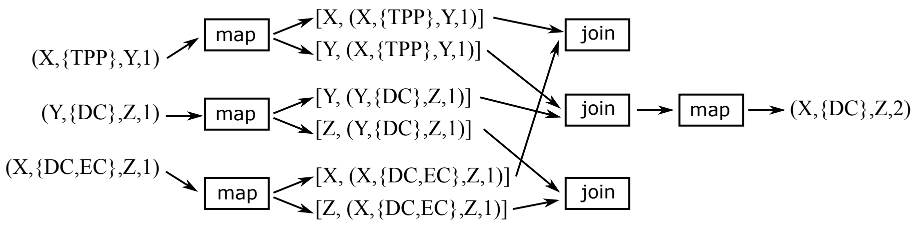

Figure 4: The Inference Stage

Inference is used to derive new relations between nodes in the constraint network. Fig.4 shows a simple example. The *map* and *join* operations are executed in parallel, allowing workload to be distributed across a number of different machines. For each edge in the dataset, a map operation returns key-value pairs where the key is a node in the network and the value is a labelled edge e.g. *[Y,(X,{TPP},Y,1)]*. Two key-value pairs are emitted for each input tuple, first with the head node as key and again with the tail node as key. Key-value pairs with common keys are then sent to the same join process. Tuples are joined when the tail node of one edge matches the head node of a second edge. In Fig.4 a join is possible between *(X,{TPP},Y,1)* and *(Y,{DC},Z,1)*. The tuples are joined on node Y. In fact, this is the only join possible. For example, it isn't possible to join *(X,{TPP},Y,1)* and *(X,{DC,EC},Z,1)* as the common node is a head node in both tuples. A map operation then takes joined pairs and derives the implied relation between nodes using a composition look-up table. Again using the example from Fig.4 there is one possible derivation:

$$(X, \{TPP\}, Y, 1) \circ (Y, \{DC\}, Z, 1) \rightarrow (X, \{DC\}, Z, 2)$$

Pseudocode for the inference stage can be seen in Algorithm 2. Smart Strategy refers to an optimisation used to prevent duplicate derivations and is explained in more detail in Section 4.6.

Algorithm 2 Inference (Smart Strategy)

```
inference(QCnetwork, i)
    //QCnetwork: A collection of edges
    //e.g. [(X,{TPP},Y,1), (Y,{TPP},Z,1), ...]
    //i: The iteration number e.g. 1
   headEdges=QCnetwork
            .filter(edge ⇒ edge.distance=$2^{i-1}$
            .map(edge ⇒ (edge.tailInterval, edge)
   tailEdges = QCnetwork
            .filter(edge ⇒ edge.distance $\leqslant 2^{i-1}$)
            .map(edge ⇒ (edge.headInterval, edge)
   joinedEdges = headEdges.join(tailEdges)
   newEdges=joinedEdges.map((key,(headEdge,tailEdge))⇒(
            headInterval = headEdge.headInterval
            tailInterval = tailEdge.tailInterval
            inferredRelation = lookUp(headEdge.relation,tailEdge.relation)
            distance = tailEdge.distance+headEdge.distance
            return (headInterval, inferredRelation, tailInterval, distance)
            ))

    return newEdges
```

4.2. The Consistency Stage

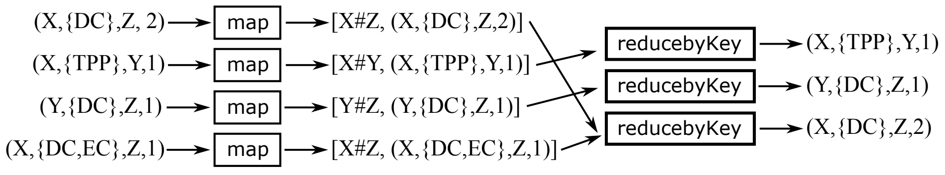

Figure 5: The Consistency Stage

Before the reasoning process begins and after new relations have been derived through the inference stage, it is necessary to check the consistency of the dataset. Fig.5 shows an example for the consistency stage of ParQR. Again *map* and *reduceByKey* operations are executed in parallel. Consistency checking involves sending all edges incident to the same two nodes in the network to the same reduce process. An edge may have originated from the existing dataset or from the inference stage. First, a map operation outputs a key-value pair for each tuple in the dataset. This time the key encodes both the head and tail nodes of the edge e.g. *Y#Z*. Key-value pairs with the same key are sent to the same reduce process. Reduction involves finding the intersection of the relations between the edges. The example in Fig.5 is a simple one, only the bottom *reduceByKey* process receives multiple tuples, *(X,{DC}Z,2)* and *(X,{DC,EC},Z,1)*, with the intersection resulting in *(X,{DC},Z,2)*. An intersection that results in an empty set indicates an inconsistency in the network that the system alerts. Algorithm 3 describes the consistency stage.

Consistency involves updating the relations between nodes. These updated relations can form the basis for further inferences. Therefore, the two stages, inference and consistency happen iteratively until no new inferences can be made or an inconsistent network is detected. Please refer back to Algorithm 1 for the main program loop.

Algorithm 3 Consistency Checking (Smart Strategy)

```
consistency(QCnetwork, i)
    //QCnetwork: A collection of edges e.g. [(X,{TPP},Y,1), (Y,{TPP},Z,1), ...]
    //i: The iteration number e.g. 1
   keyedEdges=QCnetwork
          .map(edge ⇒ (edge.tailInterval+'#'+edge.headInterval, edge))

   consistentEdges=keyedEdges.reduceByKey((edgeA,edgeB)⇒
          head = edgeA.headInterval
          tail = edgeA.tailInterval
          intersect = edgeA.relation ∩ edgeB.relation
          if |intersect| = 0
              //inconsistency detected
              stop()
          end if
          if edgeA.distance=$2^i$ and |edgeB.relation| > |intersect|
              distance = edgeA.distance
          else if edgeB.distance=$2^i$ and |edgeA.relation| > |intersect|
              distance = edgeB.distance
          else
              distance = Math.min(edgeA.distance,edgeB.distance)
          end if
          return (head, intersect, tail, distance)

    return consistentEdges
```

4.3. Generate Transpose

Before reasoning begins the transpose of the initial network is added. This is to ensure that all possible inferences can be made. For example, consider the simple constraint network *[(X,$\{R_{XY}\}$, Y),(Z,$\{R_{ZY}\}$, Y)]*. It isn't possible to infer a relation between the nodes X and Z as joins are made between the tail node of one edge and the head node of another edge. After adding the inverse of each relation i.e. *(Y,$\{R_{YX}\}$,X)* and *(Y,$\{R_{YZ}\}$,Z)*, the composition operation is then possible e.g. *(X,$\{R_{XY}\}$, Y) $\circ$ (Y,$\{R_{YZ}\}$,Z) $\rightarrow$ (X,$\{R_{XZ}\}$,Z)*. Therefore the transpose of the initial constraint network is added as a pre-processing step.

4.4. Generate Look-up Tables

The ParQR implementation involves the use of three operations on relations. The inverse of all relations is added as a pre-processing step. Composition takes place during the inference stage to derive new relations, and intersection takes place during consistency checking. Based on the input datasets, ParQR computes the results of these operations in advance. Specifically, the input dataset is analysed to identify the relations present. The composition of all possible combinations of these input relations is then calculated, along with all possible intersections between these relations. These calculations are repeated for the relations resulting from these operations, and continue iteratively until no new relations are identified. At the end of this process tables consisting of all possible composition and intersection results that could arise in the ensuing path consistency implementation are generated. These look-up tables are distributed to all machines in the computing cluster prior to the start of reasoning.

In the worst case we generate a table with $2 \cdot 2^{|\mathcal{B}|-1}$ entries for every possible composition of relations. However, ParQR only really has value as a reasoner for working with tractable subsets of binary qualitative constraint calculi. Additionally, in many large scale real world networks, the number of distinct relations present in the input dataset is often fairly small.

This pre-computation offers a number of performance advantages. First, operations, particularly composition, are simplified. In a naive approach to composition, results are computed through splitting disjunctive relations into atomic symbols, and then taking the union of the composition of each pair of atomic relations. This computation is implemented using a nested loop. Through pre-computing, composition is simplified to a single table look-up. ParQR isn't alone in pre-computing to improve performance. Hogge's method [30] [10] is a long standing approach that involves splitting each relation into two parts, performing composition using the constituent parts and then taking the union of the results. Consequently composition is reduced to four operations. GQR can also be run to pre-compute composition tables, either a full pre-computation or using Hogge's method [14].

Second, because the composition and intersection operations are pre-computed and involve looking up single values in a table, it isn't necessary to represent relation symbols in a collection class. This means relations, including disjunctive ones can be represented as numeric ids. Big data applications often have significant memory requirements. One way to reduce memory usage is to use a data representation that relies on primitive types. Typical path consistency implementations would use a collection class e.g. a bit array to represent relation symbols in a QCN. ParQR uses numeric ids to represent, not only nodes in the network, but relations as well, providing optimised memory usage.

4.5. Partial network

Unlike many existing QCN reasoners, ParQR doesn't maintain an entire network in memory. Instead, only relations where some information is known (i.e. the relation is not $\mathcal{B}$) form the input dataset. During inference, if the composition of two relations results in $\mathcal{B}$, then this information isn't added to the dataset. This also serves the purpose of reducing the memory requirements for the application. However, in the case of dense networks, where it is possible to infer useful information between many nodes in the network, the memory requirements are still a limiting factor.

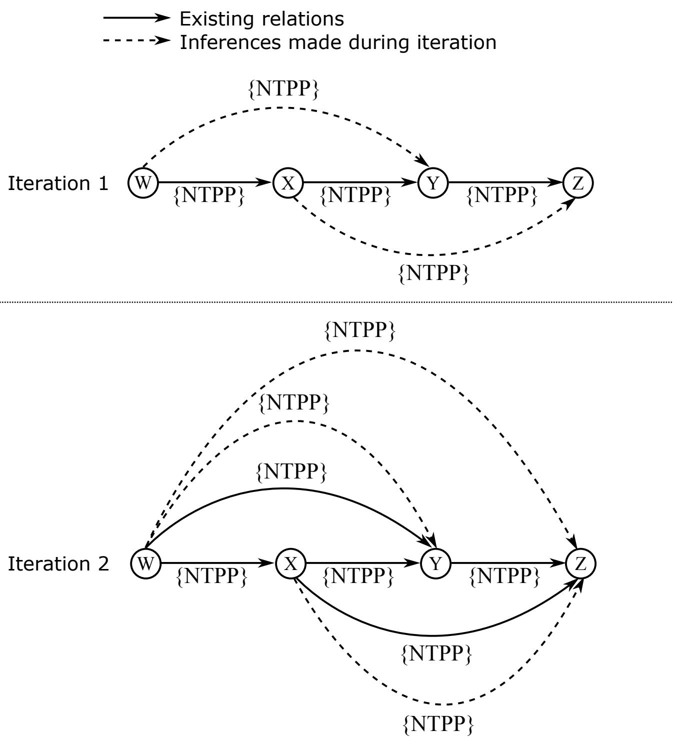

Figure 6: Derivation of Duplicate Relations in a Naive Implementation

One problem with a naive implementation suggested by Fig.4 is the derivation of duplicate relations. At each iteration, the inference stage executes joins between all edges with matching nodes, including joins that were performed in a previous iteration. Consequently at each iteration, the size of the joined dataset gets larger and larger, with many of the joins simply being duplicates of those performed at earlier rounds. See Fig.6 for an example. Join size is often a performance bottleneck in distributed applications as data needs to be shuffled through the cluster so that items with a common key can be processed at the same machine. In a naive implementation join size quickly becomes a limiting factor for the scalability of the reasoner.

Strategies for limiting duplicate derivations in a parallel, distributed environment similar to those used in ParQR have been studied previously e.g. Afrati &

Ullman [31]. They describe a property of certain distributed algorithms called unique decomposition. In algorithms that exhibit unique decomposition, the path between two nodes is only discovered once. In the case of ParQR, the join between two labelled edges is only executed once. For example, in Fig. 6 ideally the join between *(X,{NTPP},Y)* and *(Y,{NTPP},Z)* would take place at iteration 1, but not at iteration 2. Duplicate joins are prevented by storing the distance between nodes as part of each tuple. This distance value can then be used to filter which tuples can take part in the join at a given iteration.

A specific example would be a Left-Linear implementation [31]. At iteration i, in the inference phase, only those tuples with a distance of i can form the left side of the join, and only edges from the initial input can form the right side of the join. The newly derived tuple is assigned a distance value that is the sum of the distances of the two joined tuples. One disadvantage of a linear algorithm is that the number of iterations necessary to make all inferences is equal to the length of the longest path in the network. Depending on the structure of the network this can be prohibitive especially when dealing with large-scale QCNs.

An alternative algorithm, that also displays the unique decomposition property, is the Smart algorithm [32]. At iteration i, Smart combines tuples with a distance value of 2^{i-1} with tuples where the distance is strictly less than 2^{i-1}. Smart has the advantage that the number of iterations is logarithmic with respect to the longest path in the network. Although in many cases this makes it preferable to a linear algorithm, in some circumstances it can be advantageous to spread the volume of generated data over a greater number of iterations, as in the Left-Linear algorithm. ParQR offers flexibility, as an input parameter can be used to specify whether a Left-Linear or Smart strategy should be used for a particular execution of the reasoner.

It is also important to note that regardless of which strategy is adopted, duplicate derivations still occur, it is simply that when using a Smart or Left-Linear strategy, duplicate derivations via the same path are eliminated. Fig.7 shows an example where the same relation between two nodes is derived multiple times, however each derivation occurs through a different path. This is especially

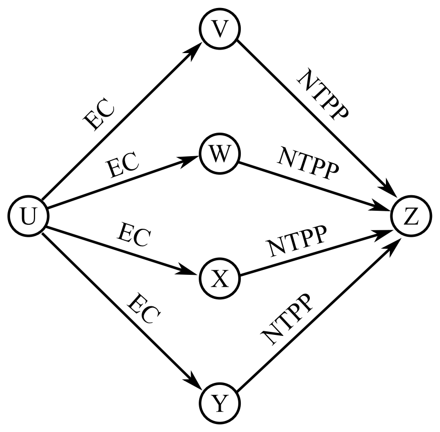

Figure 7: Duplicate Derivations

a problem in dense networks, where at points, so much data is generated that the memory limits of the application are exceeded.

Previous, work on recursive algorithms in a distributed environment such as that by Afredi & Ullman has focussed on reachability algorithms such as transitive closure. A feature of these algorithms is that even when duplicate derivations do occur (because of the different paths between two nodes) these duplicates are removed via a set difference operation. These algorithms are only interested whether it is possible, for some condition, to reach one node from another, not the actual path taken. An implementation of path consistency for QSTR is somewhat different, as different paths can yield different inferences.

Consequently, consideration needs to be given to how to deal with relations derived using different paths between the same two nodes. Removing duplicates is the responsibility of the consistency phase. All edges between the same two nodes are sent to the same reduce process, where the intersection of the relations between two nodes is used to refine relations and check for inconsistencies. At this point a decision needs to be made whether or not to allow the resulting refined edge to form the basis for inference in the following round. In the case of the Smart algorithm if at iteration i a new inference has been made resulting in a path distance of 2^i between two nodes, and the cardinality of this newly inferred relation is less than the cardinality of the existing relation between the two nodes, then it can form the basis for new inferences. The path distance value of the resulting tuple is updated to be 2^i. In the case where the cardinality is the same or greater than previously derived relations i.e. it isn't a refinement, there is no value in this tuple being used in further rounds as it will only result in exactly the same inferences as those made previously. Consequently, the distance value of the tuple is updated to be less than 2^i. See Algorithm 3 for full details. The case of using a Left-Linear algorithm is a little different, but follows the same principles, of only keeping the tuples from which new information can be derived.

5. Experiments

We have run experiments to test the capabilities and limitations of ParQR. The main purpose of running the experiments was to test the ability of ParQR to deal with large-scale qualitative constraint networks. A number of experiments were conducted involving the use of both synthetic and real world datasets. Synthetic datasets allowed us to measure the scalability of our implementation by generating input networks of varying sizes. Real world knowledge graphs were used to provide realistic examples. As described previously, path consistency is only sound and complete for tractable subsets of a binary constraint calculus. Therefore in all experiments we have used subsets of a calculus. For the synthetic datasets we choose to use Interval Algebra as the calculus and the *ORD-Horn subset*. Real world datasets were spatial in nature, and each featured relations from the tractable $\mathcal{H}_8$ subset.

5.1. Platform

The experiments were carried out using Google's Cloud Dataproc platform [8], a service that allows users to create cloud based computing clusters. A cloud based service such as Dataproc does have some disadvantages such as

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

virtualised hardware. However, cloud based services are widely used, can be easily scaled, and provide a typical real-world implementation. For the majority of the experiments a cluster consisting of 16 machines was used, each with 8 virtual CPUs and 52GB of memory. Experiment 5 involved investigating the effect of increased computing resources. In this experiment the number of machines was varied. For all the experiments reasoning times were limited to 1 hour.

5.2. Synthetic QCNs

Table 2: Dataset Characteristics for Experiments Involving Synthetic Datasets

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

Previous work evaluating the performance of spatio-temporal reasoners has commonly used randomly generated QCNs. Van Beek and Manchak [12] describe the *S(n,p)* model as a way of generating random instances and evaluating reasoners, this model was adopted by later research [33] [14] and called the *H*-model. The *H*-model is defined as *H(n,d,l)* where n is the number of nodes in a network, d the average degree of nodes and l the average label size. Label size is the number of relation symbols present in a relation e.g. the relation $X\{PO,EC\}Y$ has a label size of 2. We have used a similar approach. For experiments 1-5, n intervals were generated with randomly selected start and end points. For each interval the atomic relation between it and a randomly selected second interval was calculated. This process was repeated to obtain the desired average degree d for the network. This results in a consistent network made up of randomly selected edges each labelled with an atomic relation. For some experiments, the label size l was varied, whereby the atomic label, was replaced with a randomly selected relation from the tractable subset, such that the label still contained the original atomic label, and the overall average label size for the network resulted in l. The resulting QCN, although less constrained due to the introduction of disjunctive relations, is still consistent. Experiments were run varying the number of nodes n, the average degree d and average label size l. Table 2 shows specific details for experiments 1-5. Experiment 5 also involved the use of synthetic datasets. However, instead of varying the attributes of the dataset the purpose was to investigate how the computing cluster size affected reasoning times. ParQR was run using the same input dataset consisting of 30 million relations on computing clusters consisting of 2, 4, 8 and 16 machines.

5.2.1. Results for Synthetic QCNs

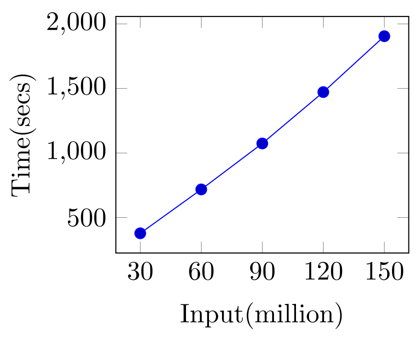

Figure 8: Experiment 1: Runtime as a function of input size on ORD-Horn IA network instances $H(n,2,1)$

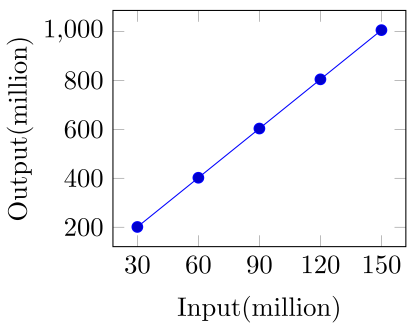

Figure 9: Experiment 1: Data volume output as a function of input size on ORD-Horn IA network instances $H(n,2,1)$

Figures 8 to 17 show the results for synthetic randomly generated uniform networks. Unless stated otherwise, the reasoner was run using the Smart strat-

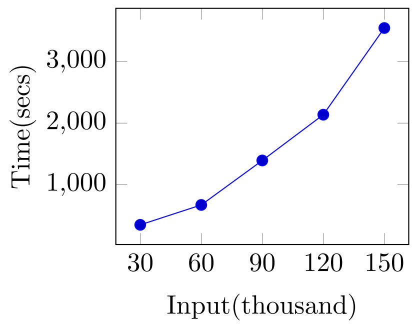

Figure 10: Experiment 2: Runtime as a function of input size on ORD-Horn IA network instances $H(n,10,1)$

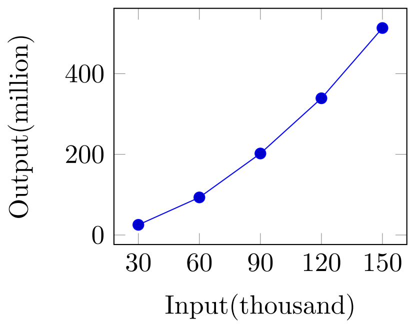

Figure 11: Experiment 2: Data volume output as a function of input size on ORD-Horn IA network instances $H(n,10,1)$

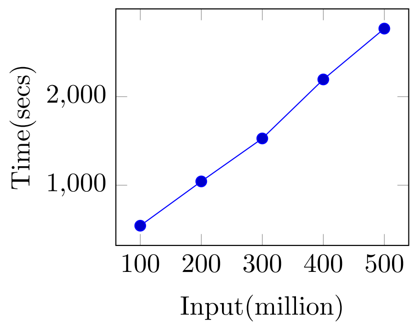

Figure 12: Experiment 3: Runtime as a function of input size on ORD-Horn IA network instances $H(n,2,6.4)$

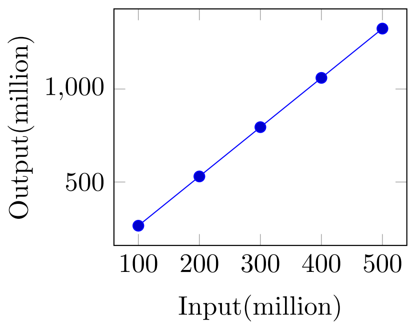

Figure 13: Experiment 3: Output as a function of input size on ORD-Horn IA network instances $H(n,2,6.4)$

egy. Fig.8 shows impressive performance, even for a comparatively small cluster, ParQR is able to handle input sizes of 150 million relations, and the runtime scales linearly. Fig.9 shows the size of the knowledge graph generated by reasoning over the inputs in experiment 1. For the large inputs, the size of the final network the reasoner is required to handle is in excess of 1 billion relations. The reasoning complexity remains constant, as the input size grows, the number of derivations grows proportionally. This is due to the comparative sparsity of the

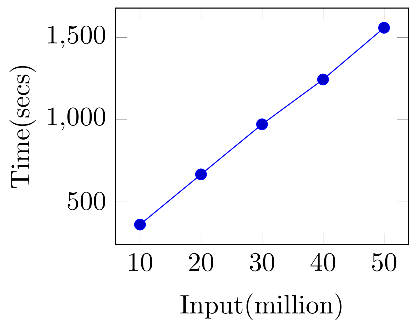

Figure 14: Experiment 4: Runtime as a function of input size on ORD-Horn IA network instances $H(n,10,6.4)$

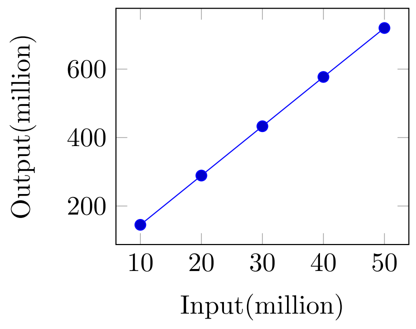

Figure 15: Experiment 4: Output as a function of input size on ORD-Horn IA network instances $H(n,10,6.4)$

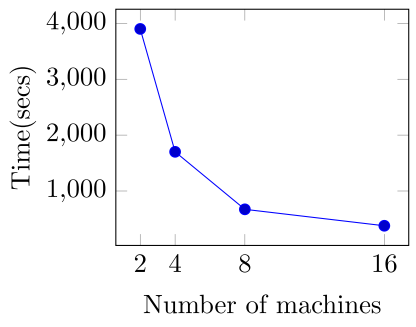

Figure 16: Experiment 5: Runtime as a function of number of machines in the computing cluster

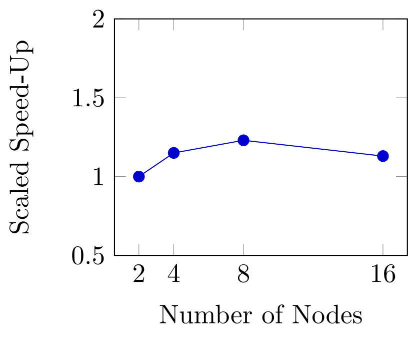

Figure 17: Experiment 5: Scaled Speed-up

networks produced in experiment 1.

The results for denser networks, Fig.10 and Fig.11, are markedly different. Here the runtimes no longer grow linearly as a function of input size. This can be explained by the volume of data generated. Fig.11 shows output approaching quadratic growth as a function of input size. This is because in the case of dense networks, it is possible to determine a relation between many more pairs of nodes e.g. for a 30,000 node network, the output is 513,209,372.

Given that the reasoner was able to handle networks of >billion relations in experiment 1, it is reasonable to question why larger networks weren't used in experiment 2, or why the runtimes aren't considerably faster. As explained above, in the case of dense networks, a relation between two nodes can be derived via many different paths. This can result in an intermediate dataset that is even bigger than a complete graph. The high data volumes occur before the removal of duplicates and reduction of multiple edges that takes place during the consistency stage. This explosion of inferences via different paths means we can view dense networks of atomic relations to be the worst case scenario for the scalability of the reasoner. For the experiments involving dense networks, a Left-Linear strategy was employed. This meant that many more iterations were needed to complete reasoning which lengthened reasoning times. However, the Left-Linear strategy was beneficial as it meant the reasoner could handle larger datasets. In the case of dense networks and the Smart strategy, so many derivations are possible within a single iteration that the memory limits of the reasoner are reached even for comparatively small networks. We attempted a similar experiment to experiment 2 using a Smart strategy but the reasoner could only handle networks up to 5,000 nodes.

Experiments 3 and 4 were repeats of experiments 1 and 2 but involved the use of a greater label size. This had a significant impact on the scale of the networks the reasoner was able to handle. In experiment 3, with an average degree of 2, the reasoner was able to reason over input datasets of 500 million relations. However, the number of derivations was very similar to that of the 150 million sized atomic network. This can be explained by the fact that the composition of disjunctive relations is more likely to result in the universal relation, $\mathcal{B}$. Universal relations aren't added to the network, as they serve no value in constraining other relations, and don't form the basis for further derivations. The same is also true for dense networks; in experiment 4 for a 10 million node network we only derive 719,908,405 relations. This is reflected in the reasoning times, which scale linearly as a function of input size.

Experiment 5 investigated the impact of computing resources on reasoning times. Unsurprisingly, as shown in Fig.16 reasoning times became much faster when larger cluster sizes were used. Fig.17 shows scaled speed-up, defined as *speed-up/number of machines in the cluster.* Ideally, scaled speed-up should be linear and remain constant with a value of one; as the number of machines are doubled, reasoning times should half. Initially, up to a cluster size of eight machines, ParQR shows better than linear performance. After this point, the system shows sub-linear speed-up. This is fairly typical for parallel distributed systems; the initial increase in computing resources makes a significant impact. However, at some point, as the cluster size gets bigger the performance gains become less dramatic. This is because, no matter how large the cluster there will always be basic time overheads such as starting jobs and distributing data.

5.2.2. Comparison with Existing Reasoners for Synthetic QCNs

Experiments 1-4 were also attempted using two existing qualitative spatial temporal reasoners, GQR [9] and Sarissa [10]. GQR is widely considered to be state of the art with regards to canonical approaches to QSTR. Sarissa was chosen because it employs triangulation specifically to tackle large scale networks. Both reasoners can provide full consistency checking; that is they can be run to employ chronological backtracking alongside path consistency. As mentioned above ParQR only uses path consistency. For this reason, both reasoners were run using path consistency only. Google's cloud service was again used to run the experiments. This time, because they are non-distributed reasoners, a single machine with 8 virtual CPUs and 52GB of memory was used. As above, reasoning times were limited to 1 hour.

[9] http://gki.informatik.uni-freiburg.de/tools/gqr/

[10] www.cril.univ-artois.fr/ sioutis/files/sarissa.tar.bz2

Table 3: Largest Datasets handled by GQR and Sarissa for Exeriments 1-4

Experiment	GQR	Sarissa
1	-	30,000,000
2	-	-
3	-	-
4	-	-

Table 3 shows the largest datasets the reasoners could return a result for. A dash indicates the reasoner was unable reason over the smallest dataset used in the experiment. Neither reasoner could handle the scale of networks ParQR was able to reason over. When faced with large networks as in experiments 1, 3 and 4, GQR reported a memory error. This is to be expected as GQR has memory requirements that are quadratic with respect to the input network size. In our experiments GQR struggled with networks bigger than 10,000 nodes in size. GQR was more successful with smaller, denser networks such as those used in experiment 2. GQR tested path consistency of a 5,000 node network with average degree of 10 in 3439 seconds. However, GQR failed to return a result within the time limit of 60 minutes for the larger networks used in experiment 2.

Sarissa performed better than GQR for sparse networks, such as those in experiments 1 and 3. Sarissa was able to test path consistency in 782 seconds for the smallest input in experiment 1, a 30 million node network, but was unable to handle any larger networks. For experiment 3, Sarissa was unable to return a result for the smallest input dataset of 100 million relations. For dense networks (experiments 2 and 4) Sarissa failed to reason over the smallest input sizes for both experiments. This isn't surprising as the triangulation approach is only really suited to sparse networks.

5.3. Real World Knowledge Graphs

In recent years a number of large scale spatial knowledge graphs have been published as linked data in RDF format. These describe topological relations between administrative areas in Europe and the world. For example, the following is an RDF triple from the gadm2 dataset [11].

```
<http://gadm.geovocab.org/id/1/47>
<http://geovocab.org/spatial#PP>
<http://gadm.geovocab.org/id/0/3> .
```

In order to reason over these datasets they need to be converted into a form understandable by reasoners. This involves dictionary encoding the triples; RDF subjects and objects are representing as integers, and predicates as RCC8 relations. For example:

```
1031 2295 (TPP NTPP)
```

The resulting datasets have then been used by researchers to evaluate reasoners e.g. [18], [34]. The characteristics of these datasets are presented in Table 4.

Table 4: Real World Knowledge Graphs

	Nodes	Edges	Avg. Degree	Avg. 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

Experiment 6 tested ParQR, GQR and Sarissa using these real-world knowledge graphs. Experiment 6 was run in the same way as experiments 1-4 i.e. the same computing resources and a limit of 1 hour on reasoning times. Two of the datasets, *gadm1* and *gadm2*, are known to be inconsistent. This is because the RCC8 relations have been computed from geometric representations, polygons

[11]http://gadm.geovocab.org/

that represent regions. Inaccuracies in the recording of the vertices for these polygons, e.g. polygons that overlap, when in fact they border each other, leads to inconsistency when path consistency is applied to the resulting RCC8 network [34]. Identifying the inconsistencies is trivial, occurring during the first iteration, therefore modified versions of these datasets with the inconsistencies removed were used for the experiments. They still provide useful examples, as the consistent versions are still large in scale and it is possible that further inconsistencies could be discovered at a later iteration.

5.3.1. Results for Real World Knowledge Graphs

Table 5: Runtime (in seconds) for Real World Knowledge Graphs

	ParQR	GQR	Sarissa
nuts	93	1.0	0.1
adm1	1338	-	395.6
gadm1	929	-	794.6
gadm2	741	-	5.8
adm2	1702	-	662.8

Table 5 shows the results for reasoning over real world networks. A dash denotes failure to complete, either because of a memory allocation error or because the time limit of 1 hour was exceeded. Both ParQR and Sarissa were able to determine path consistency for all the datasets within 1 hour. GQR was unable to fit *gadm2* or *adm2* in memory. It reasoned over *nuts* efficiently, and failed to complete within the time limit for *adm1* or *gadm1*.

Sarissa shows the best performance for all the input datasets. In some cases e.g. *nuts* and *gadm2* Sarissa was able to check path consistency in less than 10 seconds when ParQR took minutes. The largest of the networks *adm2* did push the capabilities of ParQR. One issue with real-world knowledge graphs is that they tend to be scale-free in structure, typified by a small number of hubs connected to many nodes, and a large number of nodes with few connections [21].

This skewed dataset can cause issues for a parallel distributed implementation as all the relations for a node in the network have to be processed at the same machine in the computing cluster, resulting in an uneven workload.

5.4. Discussion

As described above ParQR is able to effectively reason over large-scale QCNs with various characteristics. It is also worth considering where ParQR fits into the landscape of existing tools for reasoning over large-scale QCNs. There are a number of comparisons that are relevant:

- *Traditional approaches to QSTR e.g GQR.* As shown above, memory requirements limit the scalability of such reasoners. In comparison, ParQR is a step forward, it is able to reason over networks consisting of millions of nodes. Even in the worst-case scenario, the reasoner has proved capable of handling networks far larger in size than current reasoners.

- *Alternative approaches to the problem of large-scale QSTR such as triangulation and graph decomposition.* Unsurprisingly, a direct performance comparison in terms of speed shows ParQR being considerably slower when reasoning over the real-world datasets (*nuts*, *adm* etc.) than Sarissa. These results, although not identical, are consistent with the results from similar experiments using Sarissa presented in [18]. The longer reasoning times for ParQR are to be expected given the start-up costs associated with distributed parallel applications and the need to move data across a network; this isn't a like-for-like comparison. Existing approaches all employ either a decomposition approach or the use of triangulation to simplify the path consistency implementation. In order to exploit these strategies, reasoners such as Sarissa are only suited to networks with particular structural properties, such as sparse, scale free networks. Sarissa struggled when faced with denser networks. The results above show that ParQR is able to reason over much larger QCNs and over networks with a wider range of characteristics. As such ParQR has utility in circumstances where decomposition and triangulation are limited.

- *Existing distributed qualitative spatial temporal reasoners.* Existing distributed approaches to QSTR were described in Section 3. They are limited in terms of the scale of the networks they able to handle - up to 10 million relations, and type of networks - sparse and easy to reason over. The optimisations included in ParQR e.g. pre-computation of composition tables and reduced memory usage have made a significant impact. ParQR outperforms these existing parallel distributed implementations. The results from experiments with synthetic datasets show ParQR is able to handle much larger networks and denser networks than the current state of the art distributed reasoners described in Section 3. Additionally, we have shown that ParQR can reason over large-scale real-world networks, something that these existing tools have been unable to do.

6. Conclusions and Future Work

In this article we have presented a parallel distributed implementation of QSTR techniques that is capable of reasoning over qualitative constraint consisting of millions of relations. The approach we have taken is generalised in that it can work with any binary qualitative constraint calculi, and features a number of optimisations that allow it to handle large scale networks. These include:

- Pre-computation of the key operations in path consistency, intersection and composition.
- Implementation of algorithms that limit the derivation of duplicate relations that would otherwise severely restrict the scalability of the reasoner.

We have evaluated our reasoner using both synthetic datasets with various characteristics, and using a number of real-world knowledge graphs. Our reasoner is able to handle large-scale networks and it outperforms existing distributed approaches to QSTR.

Non-distributed approaches to reasoning over large scale networks provide superior performance on specific classes of networks. They do so by employing either triangulation or decomposition, meaning that they are suited to networks with specific structural characteristics, while our approach is suitable for any network topology.

Future work will look at the application of large-scale distributed QSTR techniques where datasets also feature quantitative data. In the case where this data is incomplete or indefinite, qualitative reasoning can be used to provide solutions and answer queries that wouldn't be possible using a purely quantitative representation.

[1] J. F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM 26 (11) (1983) 832–843.
URL `http://doi.acm.org/10.1145/182.358434`

[2] D. A. Randell, Z. Cui, A. 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, October 25-29, 1992., 1992, pp. 165–176.

[3] Y. Iwasaki, Real-world applications of qualitative reasoning, IEEE Expert 12 (3) (1997) 16–21.

[4] R. W. Focke, L. O. Wabeke, J. P. de Villiers, M. R. Inggs, Implementing interval algebra to schedule mechanically scanned multistatic radars, in: Proceedings of the 14th International Conference on Information Fusion, FUSION 2011, Chicago, Illinois, USA, July 5-8, 2011, 2011, pp. 1–7.
URL `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5977496`

[5] A. G. Cohn, S. M. Hazarika, Qualitative spatial representation and reasoning: An overview, Fundamenta informaticae 46 (1-2) (2001) 1–29.

[6] J. Renz, Qualitative Spatial Reasoning with Topological Information, Vol. 2293 of Lecture Notes in Computer Science, Springer, 2002.
URL `https://doi.org/10.1007/3-540-70736-0`

[7] M. B. Vilain, H. A. Kautz, Constraint propagation algorithms for temporal reasoning, in: Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, August 11-15, 1986. Volume 1: Science., 1986, pp. 377–382.
URL `http://www.aaai.org/Library/AAAI/1986/aaai86-063.php`

[8] J. Renz, B. Nebel, Qualitative spatial reasoning using constraint calculi, in: Handbook of Spatial Logics, 2007, pp. 161–215.
URL `https://doi.org/10.1007/978-1-4020-5587-4_4`

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010, 2010.

[10] P. B. Ladkin, A. Reinefeld, Fast algebraic methods for interval constraint problems, Ann. Math. Artif. Intell. 19 (3-4) (1997) 383–411. `doi:10.1023/A:1018968024833`.
URL `https://doi.org/10.1023/A:1018968024833`

[11] B. Nebel, Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ord-horn class, in: 12th European Conference on Artificial Intelligence, Budapest, Hungary, August 11-16, 1996, Proceedings, 1996, pp. 38–42.

[12] P. van Beek, D. W. Manchak, The design and experimental analysis of algorithms for temporal reasoning, CoRR cs.AI/9601101.
URL `http://arxiv.org/abs/cs.AI/9601101`

[13] M. Westphal, J. Hué, Nogoods in qualitative constraint-based reasoning, in: KI 2012: Advances in Artificial Intelligence - 35th Annual German Con-

ference on AI, Saarbrücken, Germany, September 24-27, 2012. Proceedings, 2012, pp. 180–192.
URL https://doi.org/10.1007/978-3-642-33347-7_16

[14] M. Westphal, Qualitative constraint-based reasoning: methods and applications, Ph.D. thesis, University of Freiburg, Freiburg im Breisgau, Germany (2015).
URL https://www.freidok.uni-freiburg.de/data/10148

[15] M. Sioutis, Y. Salhi, J. Condotta, Studying the use and effect of graph decomposition in qualitative spatial and temporal reasoning, Knowledge Eng. Review 32 (2016) e4.
URL https://doi.org/10.1017/S026988891600014X

[16] C. Nikolaou, M. Koubarakis, Querying incomplete information in RDF with SPARQL, Artif. Intell. 237 (2016) 138–171.
URL https://doi.org/10.1016/j.artint.2016.04.005

[17] M. Koubarakis, K. Kyzirakos, M. Karpathiotakis, C. Nikolaou, M. Sioutis, S. Vassos, D. Michail, T. Herekakis, C. Kontoes, I. Papoutsis, 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.

[18] M. 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, 2014, pp. 194–201.
URL https://doi.org/10.1109/ICTAI.2014.37

[19] C. Bliek, D. 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, 1999, pp. 456–461.
URL http://ijcai.org/Proceedings/99-1/Papers/066.pdf

[20] M. Sioutis, M. 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, 2012, pp. 436–443. doi:10.1109/ICTAI.2012.66.
URL https://doi.org/10.1109/ICTAI.2012.66

[21] M. Sioutis, J. 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

[22] M. Yannakakis, Computing the minimum fill-in is np-complete, SIAM Journal on Algebraic Discrete Methods 2 (1) (1981) 77–79.

[23] J. J. Li, J. Huang, J. Renz, A divide-and-conquer approach for solving interval algebra networks, in: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009, 2009, pp. 572–577.
URL http://ijcai.org/Proceedings/09/Papers/101.pdf

[24] C. Lutz, M. Milicic, A tableau algorithm for description logics with concrete domains and general tboxes, J. Autom. Reasoning 38 (1-3) (2007) 227–259. doi:10.1007/s10817-006-9049-7.
URL https://doi.org/10.1007/s10817-006-9049-7

[25] J. Huang, J. J. Li, J. Renz, Decomposition and tractability in qualitative spatial and temporal reasoning, Artif. Intell. 195 (2013) 140–164. doi:10.1016/j.artint.2012.09.009.
URL https://doi.org/10.1016/j.artint.2012.09.009

[26] M. Mantle, S. Batsakis, G. 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, 2016, pp. 29–41.
URL `https://doi.org/10.1007/978-3-319-62428-0_3`

[27] S. Nam, I. 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`

[28] J. Kim, I. Kim, Scalable distributed temporal reasoning, in: J. J. Park, V. Loia, G. Yi, Y. Sung (Eds.), Advances in Computer Science and Ubiquitous Computing, Springer Singapore, Singapore, 2018, pp. 829–835.

[29] S. Batsakis, I. Tachmazidis, G. Antoniou, Representing time and space for the semantic web, International Journal on Artificial Intelligence Tools 26 (3) (2017) 1–30.
URL `https://doi.org/10.1142/S0218213017600156`

[30] J. C. Hogge, TPLAN: A temporal interval-based planner with novel extensions, University of Illinois at Urbana-Champaign, Department of Computer Science, 1987.

[31] F. N. Afrati, J. 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`

[32] Y. E. Ioannidis, On the computation of the transitive closure of relational operators, in: VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings., 1986, pp. 403–411.
URL `http://www.vldb.org/conf/1986/P403.PDF`

[33] J. Renz, B. Nebel, Efficient methods for qualitative spatial reasoning, J. Artif. Intell. Res. 15 (2001) 289–318.
URL `https://doi.org/10.1613/jair.872`

[34] C. Nikolaou, M. 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., 2014, pp. 2724–2730.
URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8234`