Geolog: Scalable Logic Programming on Spatial Data

Tobias Grubenmann and Jens Lehmann
SDA Research Group, Department of Computer Science, University of Bonn, Germany
{grubenmann, jens.lehmann}@cs.uni-bonn.de
Fraunhofer IAIS, Dresden, Germany
jens.lehmann@iais.fraunhofer.de

Spatial data is ubiquitous in our data-driven society. The Logic Programming community has been investigating the use of spatial data in different settings. Despite the success of this research, the Geographic Information System (GIS) community has rarely made use of these new approaches. This has mainly two reasons. First, there is a lack of tools that tightly integrate logical reasoning into state-of-the-art GIS software. Second, the scalability of solutions has often not been tested and hence, some solutions might work on toy examples but do not scale well to real-world settings. The two main contributions of this paper are (1) the Relation Based Programming paradigm, expressing rules on relations instead of individual entities, and (2) Geolog, a tool for spatio-logical reasoning that can be installed on top of ArcMap, which is an industry standard GIS. We evaluate our new Relation Based Programming paradigm in four real-world scenarios and show that up to two orders of magnitude in performance gain can be achieved compared to the prevalent Entity Based Programming paradigm.

1 Introduction

Spatial data describes entities that have a *shape*, in addition to other attributes. A shape is a point, line, or polygon in a two- or three-dimensional space. We call entities that have a shape *spatial entities*. What makes spatial data unique is that many relationships between the spatial entities are implicitly given by their shape. On the one hand, this means that spatial data is very rich in implicit information. For example, given two points, the distance between those points does not need to be stored explicitly in a data structure but instead, it is sufficient to store the coordinates of these points. The distance can be computed whenever it is needed. On the other hand, the implicit nature of most spatial relations also means that these relationships have to be computed on-the-fly, which can potentially require substantial computational resources. We call the process of eliciting implicit relationships between spatial entities *spatial reasoning*.

For more complex spatial relationships, spatial reasoning alone might not be enough to infer the knowledge required to answer a question. In such cases, logical reasoning and spatial reasoning can be combined to answer the question. We call these interweaving of logical and spatial reasoning *spatio-logical reasoning*.

Spatio-logical reasoning has been popular in the last two decades. For example, in the Semantic Web community, different solutions for reasoning over spatial data have been proposed [3, 12, 18]. Meanwhile, in the Logic Programming (LP) community, spatial extensions for LP [8, 13] have been explored.

What all these approaches have in common is that spatial entities are represented as individual objects and relationships are established between these objects. This is a very natural way of modelling spatial

Submitted to:
ICLP 2021

entities in a reasoning system, as the rules governing the relationships between them can be expressed on the level of individual spatial entities. We call this approach the *Entity Based Programming* paradigm. This approach works especially well when there is only a single spatial operation required. However, subsequent spatial operations have to be performed for each resulting object of the first spatial operation.

There is, however, an alternative way to model spatial entities by using relations. Many spatial operations can work with relations both as input and output. Therefore, we propose a *Relation Based Programming* paradigm, in which spatial operations create relations as outputs, which can be used as input for subsequent spatial operations. Hence, our research hypothesis is that *passing references to relations is more efficient than iterating over individual objects*. Indeed, as we show in the evaluation section, this is the case for different scenarios. Thus, we propose to drop the restriction that reasoning rules on spatial entities should be expressed on individual entities and instead, we show that for scalability reasons, it is often more efficient to express certain reasoning rules on relations. Hence, the problem statement of this work can be described as: *Can the scalability of spatio-logical reasoning be improved by expressing reasoning rules on relations instead of entities?*

In this work, we present (1) the *Relation Based Programming* paradigm, which expresses reasoning rules on spatial data using relations, (2) *Geolog*, a new reasoning plugin for ArcMap based on Logic Programming enabling Relation Based Programming in addition to Entity Based Programming, and (3) a comparison of the scalability of the different programming paradigms. ArcMap (a.k.a ArcGIS Desktop) is a GIS application that is considered an industry standard [11]. At its core, Geolog provides the ability to use the ArcMap API within Prolog. For this, Geolog integrates the arcpy library—a Python API for ArcMap—and PySwip—a Python interface to SWI Prolog—and thus, provides the ability to use many spatial functions that are provided by ArcMap.

Finally, to enable GIS users to make use of the reasoning power of any reasoning system, it is important to provide the means to integrate the reasoner into existing GIS software and GIS workflows. Spatial data is visual by nature and hence, it is desirable that the output of the reasoning process can be seamlessly integrated into GIS software that is designed to visualize and further process the spatial data. Geolog achieves this via side-effects of certain predicates. For example, one side-effect could be that a predicate creates a new selection within an ArcMap map document (Figure 1). Such a new selection allows the GIS user to directly continue with these newly selected entities within or outside of Geolog. We implement Geolog as an ArcMap Python-Addin which can be easily installed on the most recent versions of ArcMap[1]. Full integration is achieved by incorporating the arcpy library into Geolog.

2 Preliminaries

Spatial operations take spatial data as input. Spatial data is any kind of data that has a *shape* (point, line string, polygon) attached to its entities. We call such entities *spatial entities*. For example, determining the distance between two Points of Interest (POIs) on a map is a spatial operation taking two spatial entities as input. Besides other information like the name of the POI or the category, each POI also has a shape, in this case a point. The spatial operation uses these shapes to define the output, i.e., the distance between the two points defining the location of the POIs. Spatial data can be very rich in information because many relationships are implicitly given by the location and orientation of different shapes. *Spatial reasoning* can be seen as the task of making these implicitly given relationships explicit.

A *spatial predicate* is a Prolog-predicate which makes use of spatial operations to determine the truth-value of the predicate or to instantiate unbound variables. For instance, let `distance(X, Y, D)` be a

[1] At the time of writing, the most recent version is ArcMap 10.8.1

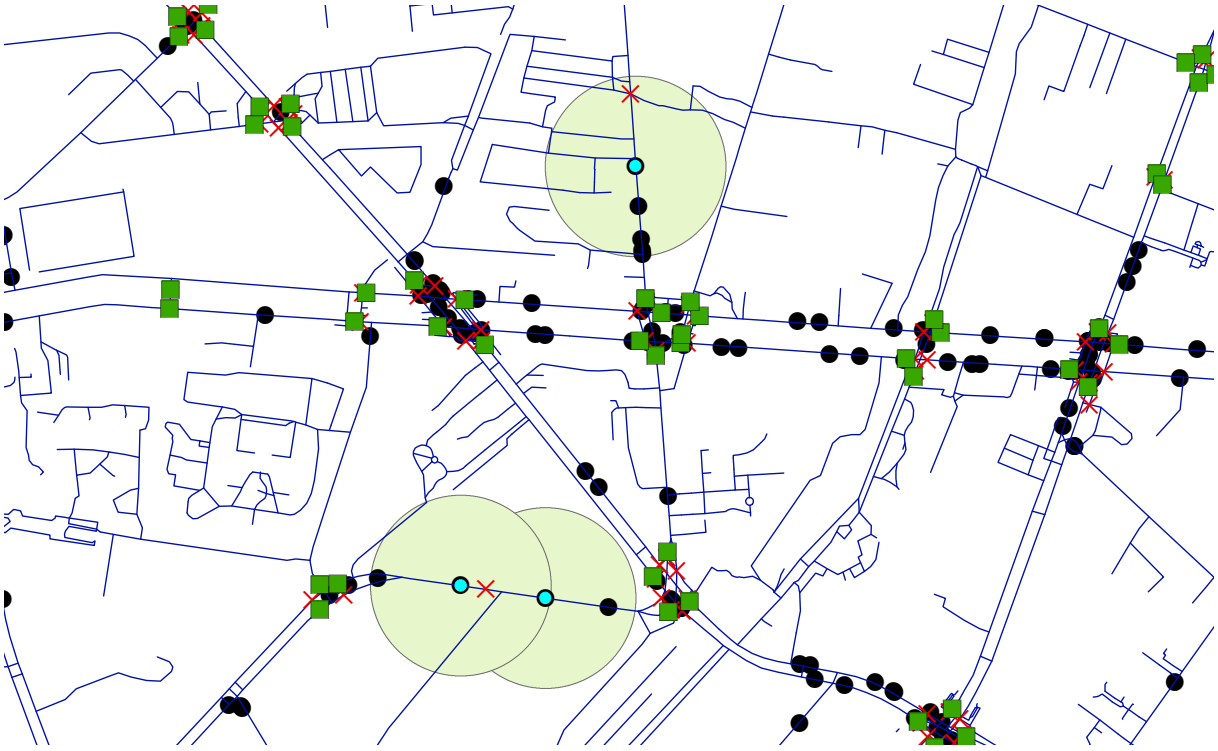

Figure 1: Selected accidents (○) near crossings (×) without traffic lights (■), and unselected accidents (●)

predicate which is true when D is the distance between two points X and Y. In this case, `distance(X, Y, D)` is a spatial predicate.

Spatio-logical reasoning incorporates logical reasoning on spatial and non-spatial entities and spatial reasoning on spatial entities. Whereas both, logical and spatial reasoning, have the same goal of making implicit facts explicit, spatial reasoning is often performed in dedicated libraries or systems. The reason for this is that (1) calculating spatial relationships is a non-trivial task requiring dedicated algorithms, (2) different coordinate systems and projections require reprojecting, and (3) spatial indices might be required for scalable solutions. However, having a dedicated library can also pose some challenges, as the calls to this library need to be interwoven into the logical reasoning process. The remainder of this section will discuss the two different approaches using the example below.

Example 1 *The goal is to find for each accident the nearby traffic entities (signals, crossings, etc.) and the streets to which these traffic entities belong.* ▲

To understand the difference between the programming paradigms mentioned in this work, we need to distinguish between a *logical layer* and a *spatial layer*. The logical layer is the part of the reasoning process that uses logical reasoning to establish new facts. In our specific implementation, the logical layer corresponds to all reasoning steps performed within Prolog. The spatial layer performs spatial operations using the underlying spatial libraries (or spatial databases). A *spatial predicate* inside the logical layer is a predicate that uses a spatial operation, directly or indirectly, in its definition. Note that a spatial predicate can use more than one spatial operation.

```
nearbyTrafficAndStreet(A, T, S) :- accident(A), near(A, T), traffic(T),
                                   close(T, S), street(S).
```

Figure 2: A predicate for the Entity Based Programming paradigm for traffic entities near accidents and their street.

```
accident(a1). traffic(t1). traffic(t2). street(s1). street(s2).
```

Figure 3: Facts assumed for the predicate in Figure 2.

Entity Based Programming Paradigm

The *Entity Based Programming* paradigm is as follows: Each spatial entity is modelled as a separate atom in the logical layer. If a call to the spatial operation has no unbound variables, the spatial operation returns true or false. If a call to the spatial operation has unbound variables, the spatial operation returns the atoms corresponding to all spatial entities matching the spatial operation in an iterative manner.

To improve the performance of the Entity Based Programming paradigm, the spatial entities can be stored in a backend storage (e.g., a spatially enabled database [12]). Using such a backend, it is possible to exploit spatial indices to quickly elicit the result set of all matching spatial entities. Once all spatial entities are found, the individual atoms are returned by iterating over the result set and matching the entity to the corresponding atom.

We will now discuss how the Entity Based Programming paradigm is used in Example 1. For this, we assume that there is a spatial predicate `close(X, Y)` which is true if the spatial entity corresponding to atom `X` is not farther away than 10 metres from the spatial entity corresponding to atom `Y`. Using `close`, we can define that an entity `X` is on a street `Y`, if they are closer than 10 metres. We further assume that there are predicates `accident(X)`, `traffic(X)`, and `street(X)`, which are true when `X` is an atom representing an accident, a traffic feature, and a street, respectively. Finally, we assume a predicate `near(X, Y)` which is true if a spatial entity denoted by atom `X` is not farther away than 100 metres from a spatial entity denoted by atom `Y`.

A possible query for this example could look like the one in Figure 2.

Let us further assume the facts in Figure 3. Let us assume that `t1` and `t2` are near `a1`, and that `s1` is close to `t1` and `s2` close to `t2`, then the predicate in Figure 2 is evaluated as follows. First, accident `a1` is retrieved on the logical layer using predicate `accident(A)`. Then, the spatial predicate is used to determine which spatial entities `T` are near `a1`. For this, a spatial operation is used which determines the result set {`t1`, `t2`}. The spatial operation returns first `t1`, which is tested by predicate `traffic(T)` whether it is a traffic entity or not. As it is the case, the spatial predicate `close(T, S)` is used next to determine the entities `S` that are close to `t1`. The result set of this second spatial operation is {`s1`} and hence, the spatial predicate returns `s1` as the only match. Finally, `s1` is tested whether it is a street or not. As it is the case, we have found the first solution for the predicate `nearbyTrafficEntityOnStreet(A, T, S)`: {`A: a1, T: t1, S: s1`}. Now, we can backtrack and retrieve the second entity matching the predicate `near(a1, T)`. The same steps are performed on `t2` to obtain the second solution: {`A: a1, T: t2, S: s2`}.

```
nearbyTrafficAndStreet(A, T, S, R) :- accidents(A), traffics(T), streets(S),
                                      near(A, T, R1), close(T, S, R2),
                                      join(R1, R2, R).
```

Figure 4: A predicate for the Relation Based Programming paradigm for traffic entities near accidents and their street.

The important observation here is that each matching traffic entity for `near(a1, T)` results in a separate call to the predicate `close(T, S)`. Also, for every call we need to send atoms corresponding to spatial entities back and forth between the logical and spatial layer.

3 Relation Based Programming Paradigm

In the following, we introduce our new Relation Based Programming Paradigm, which complements the Entity Based Programming Paradigm. Note that both programming paradigms can be mixed within a single Logic Program by, either, iterating over a relation to get single entities from a relation, or, collecting single entities and store them in a new relation.

The *Relation Based Programming* paradigm is as follows: *Relations* are used to represent sets of entities and relationships between entities. These relations are represented on the logical layer as specific atoms, where a single atom in the logical layer can correspond to a relation consisting of an arbitrary number of entities. Correspondingly, the spatial predicates take as input atoms representing those relations rather than individual entities. In practice, such relations can be realized as database relations. We will denote relations which store a collection of entities as *entity-relation* and relations which store relationships between different entities as *relationship-relation*. Note that entity-relations contain all the attributes associated to a spatial entity, including its shape. In contrast, a relationship-relation only contains the foreign keys which point to the entities in an entity-relation for which the relationship is true.

To understand how the *Relation Based Programming* paradigm works, we come back to Example 1. Figure 4 illustrates how the predicate for Example 1 could be defined. Most notably, the predicate has now a fourth parameter R which represents the relationship-relation that will be produced by the spatial operation. As a first step, A, T, and S will be bound to the entity-relations containing all accidents, traffic entities, and streets, respectively. Afterwards, the `near(A, T, R1)` relation binds $R1$ to the relationship-relation of all pairs of accidents that are near traffic entities. Next, the `close(T, S, R2)` relation binds $R2$ to the relationship-relation of all pairs of traffic entities that are close to streets. Finally, a join between $R1$ and $R2$ yields the relationship-relation R that contains all triples of accidents, traffic entities, and streets such that the traffic entity is near the accident and the street is close to the traffic entity.

4 Programming with the Relation Based Programming Strategy

In the following, we discuss the different predicates that we use in the evaluation section for both the Entity Based Programming paradigm and Relation Based Programming paradigm.

For the Relation Based Programming paradigm, we distinguish between *entity-relations*, which contain spatial entities with the associated attributes including the shape, and *relationship-relations*, which contain tuples of foreign keys but no shape.

Entities are uniquely identified by a pair of the form `(category, ID)`, e.g., `("accidents", 1)`. Relations are uniquely identified by the name of the relation, e.g., `"accidents"`.

4.1 Spatial Predicates

Spatial predicates are naturally an important part of spatio-logical reasoning. For the Entity Based Programming paradigm, spatial predicates are defined the following way:

Definition 1 (Entity-based near predicate) *The predicate `near(E_1, E_2)` is true if the distance between the entities `E_1` and `E_2` is not larger than 100 metres.* ▲

The predicate `closeby(E_1, E_2)` is defined analogously, with a threshold of 10 metres instead of 100 metres.

For the Relation Based Programming paradigm, spatial predicates are defined the following way:

Definition 2 (Relation-based near predicate) *The predicate `near_relational(E_1, E_2, R, [A_1, A_2])` is true if `R` is a relationship-relation whith attributes `A_1` and `A_2` which contain all keys from entity-relations `E_1` and `E_2`, respectively, such that the spatial entities corresponding to those keys are not farther away than 100 metres.* ▲

The predicate `closeby_relational(E_1, E_2, R, [A_1, A_2])` is defined analogue, with a threshold of 10 metres instead of 100 metres.

Given such a relation-based spatial predicate, it is often useful to restrict the further reasoning process on those entities that satisfy a given spatial predicate. We can use the identifiers in the relationship-relation to filter for those points:

Definition 3 (Relation-based filter predicate) *The predicate `filter_by_relationship(E_IN, R, A, E_OUT)` is true if the entity-relation `E_OUT` contains all spatial entities which are in entity-relation `E_IN` and their key is contained in the attribute `A` of relationship-relation `R`.* ▲

4.2 OSM-Specific Predicates

The following predicates are specific for identifying different types of entities, as defined by OpenStreetMap (OSM) [15]. OSM uses a 4-digit code to determine the type of a point, line, or polygon. In addition to specific types, OSM also defines code ranges for supertypes. E.g., schools have the code 2082, whereas entities related to education have an OSM code between 2080 and 2089. Using these codes, it is possible to define rules which apply only to certain types of entities. The following two predicates are used to determine or test the type of an entity.

Definition 4 (Entity-based type predicate) *The predicate `entity_type(T, E)` is true if entity `E` is of type `T`.* ▲

Definition 5 (Relation-based type predicate) *The predicate `entity_type_relational(T, E_IN, E_OUT)` is true if entity-relation `E_OUT` contains all entities in entity-relation `E_IN` which are of type `T`.* ▲

4.3 Relational-Specific Predicates

The following three predicates are only used when working with relations.

Definition 6 (Relation-based join predicate) *The predicate* `join_relational(E_1, E_2, E_OUT, A)` *is true if the relationship-relation* `E_OUT` *contains all pairs of entities from relationship-relations* `E_1` *and* `E_2` *which have the same value on attribute* `A`*.* ▲

The projection can be helpful to remove attributes from a relationship-relation which are not needed anymore.

Definition 7 (Relation-based projection) *The predicate* `project_id_relational(E_in, L, E_OUT)` *is true if relationship-relation* `E_OUT` *contains all attributes in list* `L` *for each record in relationship-relation* `E_IN`*.* ▲

Finally, the relation-based difference is defined as follows:

Definition 8 (Relation-based difference) *The predicate* `minus_relational(E_1, E_2, E_OUT)` *is true if relationship-relation* `E_OUT` *contains all entities in relationship-relation* `E_1` *that are not in relationship-relation* `E_2`*.* `E_1` *and* `E_2` *must have the same attributes.* ▲

5 The Geolog Tool

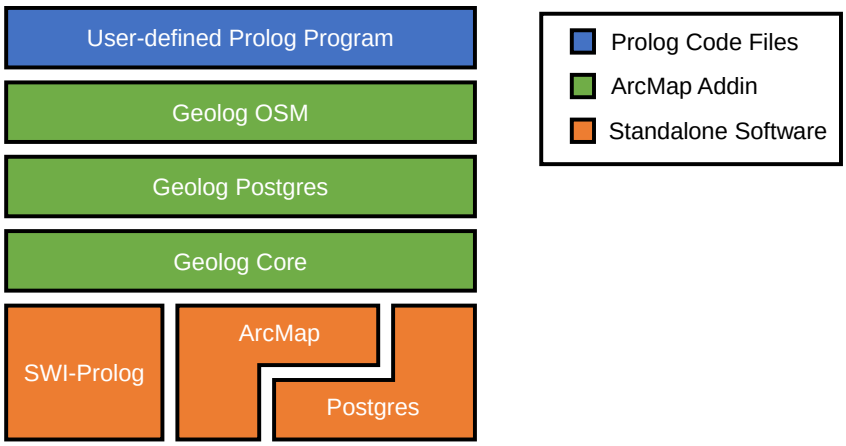

Figure 5: The different layers of Geolog.

In Figure 5, we illustrate the different layers of our Geolog approach. At the bottom, in orange, we have SWI-Prolog, ArcMap, and Postgres as standalone software components. Note that the access to Postgres can be via ArcMap or directly from the Geolog core.

Above the standalone software, in green, is the *ArcMap Addin*, which consists of three different components. Addins are a plugin concept for ArcMap and allow easy installation of tools and toolbars with simple UI elements. The first component, *Geolog Core*, provides core functionalities which are required to communicate with ArcMap. This component also handles communication with the Prolog interpreter and keeps track of which Prolog atom corresponds to which Python object. The next component, *Geolog*

Postgres, provides predicates which facilitates the communication with the database. All predicates from Section 4, except the OSM specific predicates, are implemented in this component. Finally, the *Geolog OSM* component provides predicates for filtering different OSM types and the atoms corresponding to these types.

The top most layer, in blue, contains the application-specific Prolog code written by the user. In Section 6, we introduce different scenarios with some examples of user code.

To communicate with ArcMap, the Geolog Core Addin uses the *arcpy* library, which provides a Python interface for most functionalities within ArcMap. Geolog Core maps 398 classes and 3058 functions from the arcpy library and makes them available to a Prolog Programm.[2] These classes and functions not only provide results that can be further processed within Geolog but they often also have side-effects which can be exploited. For example, it is possible to manipulate a map object using arcpy to illustrate the result of the program or to create data structures for further processing. In Figure 1, we have created a map that illustrates all accidents near crossings without traffic lights. For this, we extended the program from Figure 17 to draw a 100 meter radius around each accident (to illustrate which objects are near), to create a new collection with all crossings, to create a new collection with all traffic lights, and to select all accidents that match our criteria (near a crossing that has no traffic light).

6 Evaluation

In the following, we discuss the scaling behaviour of four different scenarios using both the Entity Based Programming (Entity in Figures 6–9) and Relation Based Programming (Relation in Figures 6–9) strategies. In addition, we also evaluate the scaling behaviour when there is a need to iterate over the result of the Relation Based Programming paradigm and thus, adding an additional iteration over the result at the end of the query (Rel. Iterator in Figures 6–9). The scaling is evaluated with respect to the number of accidents. For this, we randomly sample a certain number of accidents from all accidents in Berlin. Each point in the plot is the average of ten runs. All scaling plots are in log-log scale.

The scenarios focus on the analysis of accidents in Berlin, Germany. For this, we use the official accident data of 2019.[3] To establish the context of the accidents, we use the OpenStreetMap data for Berlin, prepared by Geofabrik.[4] For the `near` predicates, a threshold of 100 metres has been chosen and a threshold of 10 metres for the `closeby` predicates.

All experiments were conducted on a PC with 16GB of RAM and a 1.6GHz 4 core processor. As software, we used ArcMap 10.8.1 (32-bit), Python 2.7.18 (32-bit), SWI-Prolog 8.2.4 (32 bit), and the SWI-Prolog-Python interface PySwip 0.2.10[5].

For the evaluation, v0.1.1 of the Geolog Addin was used, available at `https://github.com/tobiasgrubenmann/geolog_addin`. The Prolog code for the evaluation is available at `https://github.com/tobiasgrubenmann/geolog_accident`.

6.1 Scenario 1: Accidents near crossings

The aim of this first scenario is to establish how the different programming paradigms perform on a simple query with one spatial operation. For this, we query for accidents near pedestrian crossings. The

[2] The reported number of classes and functions is for ArcMap 10.8.1 with Advanced License. The number might differ depending on the version and license.

[3] `https://unfallatlas.statistikportal.de/_opendata2020.html`

[4] `http://download.geofabrik.de/`

[5] `https://github.com/yuce/pyswip`

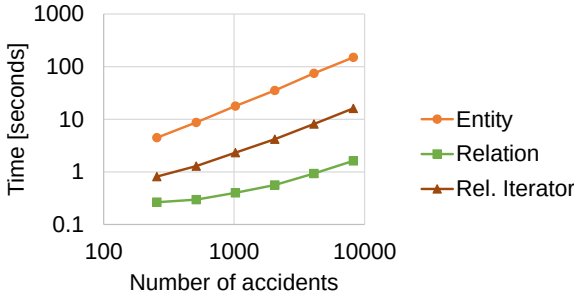

Figure 6: Scaling for Scenario 1

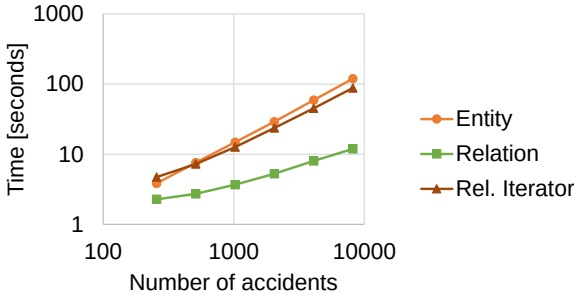

Figure 7: Scaling Scenario 2

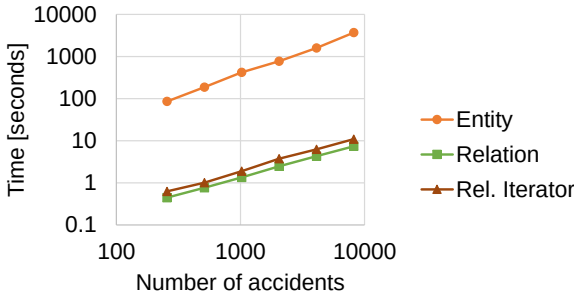

Figure 8: Scaling Scenario 3

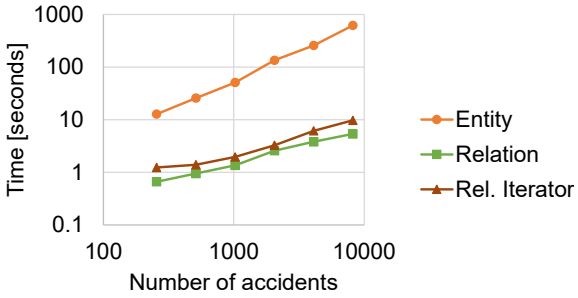

Figure 9: Scaling for Scenario 4

```
:- near(("accidents", AccidentID), ("traffic", TrafficID)),
   entity_type(crossing_features, ("traffic", TrafficID)).
```

Figure 10: Query for Scenario 1 using the Entity Based Programming paradigm

two queries for the Entity Based Programming (Figure 10) and Relation Based Programming (Figure 11) both use one spatial predicate and one type predicate. The scaling is shown in Figure 6. As we can see from the figure, all three approaches have the same scaling behaviour. However, the execution times are roughly one order of magnitude apart from each other. For example, to analyse 8192 accidents, the Entity Based Programming requires around 150 seconds, the Relation Based Programming with an iterator at the end requires only 16 seconds, and the Relation Based Programming without iterator requires only 1.6 seconds for the same query.

6.2 Scenario 2: Traffic entities on the same street as accidents

The aim of this scenario is to test the scaling behaviour when two spatial operations are involved which have to be chained after each other. For this, we query for accidents and traffic features that are on the same road. For the Entity Based Programming, we just chained two spatial predicates after each other (Figure 12). In contrast, the Relation Based Programming (Figure 13) has an additional filter predicate to restrict further processing to those roads that are close to accidents. Also, the Relation

```
:- entity_type_relational(crossing_features, "traffic", Crossings),
   near_relational("accidents", Crossings, Result).
```

Figure 11: Query for Scenario 1 using the Relation Based Programming paradigm

```
:- closeby(("accidents", AccidentID), ("roads", RoadID)),
   closeby(("traffic", TrafficID), ("roads", RoadID)).
```

Figure 12: Query for Scenario 2 using the Entity Based Programming paradigm

Based Programming requires a final join to retrieve all accidents that share the same road as the traffic entities.

As one can see in Figure 7, Entity and Rel. Iterator are close together. This can be explained by the fact that the number of intermediate results (streets) is considerably smaller than the end result (traffic entities and their streets for each accident). Therefore, the Relation Based Programming is only beneficial when there is no need to iterate over the large end result. In contrast, Relation is one order of magnitude faster than Entity and in addition has a flatter scaling curve.

6.3 Scenario 3: Accidents near POIs that are near schools

The third scenario queries for accidents that are near points of interest (POIs) that are near schools. The scenario is similar to the last one, in the sense that both are chaining two spatial predicates. However, this time, the intermediate results will be much larger since there are many POIs on the map. Indeed, as Figure 8 shows, Rel. Iterator is very close to Relation. In addition, both outperform Entity by more than two orders of magnitude. For example, for 8192 accidents, Rel. Iterator and Relation require around 10 seconds, whereas Entity requires almost an hour to get the same result.

The two queries for this scenario are shown in Figures 14 and 15.

6.4 Scenario 4: Accidents near crossings without traffic lights

The final scenarios are testing the scaling behaviour when querying with negation. We are looking for accidents that happened at crossings that have no traffic lights nearby. In the Entity Based Programming paradigm, we use the negation `\+` to test for the absence of traffic lights (Figure 16). For the Relation Based Programming paradigm (Figure 17), we make use of the `minus_relation` predicate to substract

```
:- closeby_relational("accidents", "roads", AccRoads, ["Acc", "Road"]),
   filter_by_relationship("roads", AccRoads, "Road", Roads),
   closeby_relational("traffic", Roads, TrafficRoads, ["Traffic", "Road"]),
   join_relational(AccRoads, TrafficRoads, Result, "Road",
      ["Acc", "rel1.Road", "Traffic"]).
```

Figure 13: Query for Scenario 2 using the Relation Based Programming paradigm

```
:- near(("accidents", AccidentID), ("pois", Pois1)),
   near(("pois", Pois1), ("pois", Pois2)),
   entity_type(school_features, ("pois", Pois2)).
```

Figure 14: Query for Scenario 3 using the Entity Based Programming paradigm

```
:- near_relational("accidents", "pois", AccidentPois, ["Acc", "Poi"]),
   filter_by_relationship("pois", AccidentPois, "poi", Pois),
   entity_type_relational(school_features, "pois", Schools),
   near_relational(Pois, Schools, PoisSchools, ["Poi", "School"]),
   join_relational(AccidentPois, PoisSchools, AccidentPoiSchool, "Poi",
      ["Acc", "rel1.Poi", "School"]).
```

Figure 15: Query for Scenario 3 using the Relation Based Programming paradigm

the pairs of accidents and crossings with traffic lights from the relation containing all pairs of accidents and crossings. The results are pairs of accidents and crossings without traffic lights. Since the `minus_relation` predicate requires the schemas of both relations to match, we also use a projection predicate.

As one can see in Figure 9, the scaling behaviour of Relation and Rel. Iterator are close. This can be explained by the fact that the end result is quite small and hence, the overhead of iterating over the end result does not have such a big impact. Entity is almost two orders of magnitude slower than the others for 8192 accidents. Moreover, the slope of the scaling curve is steeper for Entity.

7 Related Work

Region Connection Calculus (RCC) [6] is used for representing and reasoning over regions. RCC describes the possible relations between two regions. Nutt [14] introduced the *topological set constraints* language as a generalization of RCC and reduced reasoning about topological constraints to reasoning in modal propositional logic S4.

The Dimensionally Extended 9-Intersection Model (DE-9IM) [5] is a model to describe topological relations that are invariant to translation, rotation, and scaling. Different relations between two objects are represented in a 3×3 capturing the intersections of interior, exterior, and boundary. This model is used in GIS applications like ArcMap and spatial databases like PostGIS to determine spatial relations between objects [7, 17].

```
:- near(("accidents", AccidentID), ("traffic", Traffic)),
   entity_type(crossing_features, ("traffic", Traffic)),
   \+(near(("traffic", Traffic), ("traffic", OtherTraffic)),
      entity_type(traffic_signal_features, ("traffic", OtherTraffic))).
```

Figure 16: Query for Scenario 4 using the Entity Based Programming paradigm

```
:- entity_type_relational(crossing_features, "traffic", Crossings),
   near_relational("accidents", Crossings, AccCrossing, ["Acc", "Crossing"]),
   filter_by_relationship("traffic", AccCrossing, "Crossing", CrossFilt),
   entity_type_relational(traffic_signal_features, "traffic", Signals),
   near_relational(CrossFilt, Signals, CrossingsSignals, ["Crossing", "Sig"]),
   join_relational(AccCrossing, CrossingsSignals, Join, "Crossing",
      ["Acc", "rel1.Cross", "Sig"]),
   project_id_relational(Join, ["Acc", "Crossing"], AccSignal),
   project_id_relational(Accrossing, ["Acc", "Crossing"], AllAcc),
   minus_relational(AllAcc, AccSignal, Result).
```

Figure 17: Query for Scenario 4 using the Relation Based Programming paradigm

The logic programming community has produced many approaches that focused on expressing spatial reasoning directly, rather than by interfacing to a GIS. Most of this work [2, 4, 16, 19] aims at pushing the boundaries of spatial reasoning to ever more challenging problem categories (e.g. non-monotonic reasoning) rather than on the scalability of the approach or on issues external to reasoning, such as visualisation of the results of the spatial reasoning process. One notable exception is the Space package for SWI-Prolog [8]. It also targets efficiency by integrating semantic and spatial indexes in SWI-Prolog. Having both types of indexes in the same system enables advanced query optimization. In general, their approach follows the Entity Based Programming paradigm, with additional optimization for the NN search.

λProlog(QS) [13] introduces a framework for spatial reasoning within high-order logic programming. The algebraic semantics of spatial relations is implemented using Constraint Handling Rules. Using such rules, λProlog(QS) can simplify terms before evaluation the spatial relationships on a numerical level. The evaluation of the spatial relationships on the numerical level uses a Entity Based Programming paradigm.

Geosparql [3] introduced a standard for representing and querying spatial data in the Semantic Web. In Geosparql, spatial features have a special predicate that points to the actual geometry, which is represented as a string. Whereas Geosparql does not specify the implementation details of the different spatial predicates, the restriction of representing geometries as individual entities implicitly favours a Entity Based Programming paradigm.

The data model *stRDF*, the query language *stSPARQL*, and the Geospatial DBMS *Strabon* together form a scalable solution to query spatial data in the Semantic Web [12]. As backend for spatial operations, the *PostGIS* extension of the *Postgres* DBMS is used. The paper does not discuss the optimization using a Relation Based Programming paradigm and so, to the best of our knowledge, only the Entity Based Programming paradigm is used.

PelletSpatial [18], which is built on top of OWL 2 reasoner Pellet, implements two Region Connection Calculus (RCC) reasoners. Since geometries are represented as individual entities, a Entity Based Programming paradigm is naturally favoured in this setting. However, specific evaluation strategies for spatial operations are not discussed in detail.

The Geographica 2 benchmark [10] evaluates different spatially enabled RDF stores. The benchmark emphasizes the scalability of the systems.

The problem of interfacing logic and databases has been extensively studied by the deductive database (DDB) community. The impedance mismatch of tuple-oriented, top-down evaluation of logic programming and set-oriented bottom-up evaluation has let, already in the early eighties, to the agreement that accessing a relational database from Prolog is unacceptably inefficient. This has sparked research on how to simulate Prolog's top-down evaluation in a bottom up, set-oriented fix-point computation. The magic-set [1] evaluation was a breakthrough step, followed by ever more sophisticated and efficient implementations. Handling recursive programs with function symbols was addressed, among others, in LogicBase [9]. However, most of the research in the DDB community focused on programs without function symbols.

Our work partly confirms the old insight about the inefficiency of tuple-oriented access to a relational (or, in our case, geographic) database. As shown in Section 6, the purely entity-based approach is indeed orders of magnitude slower. However, we also showed that, in spite of its tuple-oriented evaluation paradigm, logic programming can be a very efficient high-level query language for databases, if used properly. The key point is to let Prolog terms represent entire relations rather than individual database entities. This way, Prolog predicates translate to database queries that exploit the power of set operations and take advantage of all the sophisticated relational and spatial indexing mechanisms of the database.

8 Conclusion

In this paper, we investigated a new paradigm for spatio-logical reasoning using relations instead of single entities as basic building block. We have illustrated how this new approach can be used in four different scenarios. As the scaling evaluation shows, a user can often save one or two orders of magnitude of execution time when switching to the Relation Based Programming approach. In addition, we also discussed the importance of a tight integration with existing GIS software. To this end, we introduced Geolog, a Python Addin for ArcMap. We have also seen that writing logical rules which operate on relations can become slightly more complicated than the more natural way of writing rules which operate on single entities. Feature work includes further studies into an automatic conversion of the latter into the former such that a user can benefit from good performance and simpler rules.

Acknowledgements

This work was partially funded by the Federal Ministry of Education and Research (BMBF), Germany under Simple-ML (01IS18054) and the European Commission under PLATOON (872592) and Cleopatra (812997). Map data copyrighted OpenStreetMap contributors and available from `https://www.openstreetmap.org`.

Special thanks to Günter Kniesel-Wünsche for his valuable insights into Logic Programming.

References

[1] François Bancilhon (1986): *Naive Evaluation of Recursively Defined Relations*. In Micheal L. Brodie & John Mylopoulos, editors: *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, Topics in Information Systems, Springer, New York, NY, pp. 165–178, doi:10.1007/978-1-4612-4980-1_17.

[2] George Baryannis, Ilias Tachmazidis, Sotiris Batsakis, Grigoris Antoniou, Mario Alviano, Timos Sellis & Pei-Wei Tsai (2018): *A Trajectory Calculus for Qualitative Spatial Reasoning Using Answer Set Programming*. *Theory and Practice of Logic Programming* 18(3-4), pp. 355–371, doi:10.1017/S147106841800011X.

[3] Robert Battle & Dave Kolas (2011): *GeoSPARQL: Enabling a Geospatial Semantic Web*. *Semantic Web*, p. 17.

[4] Mehul Bhatt, Jae Hee Lee & Carl Schultz (2011): *CLP(QS): A Declarative Spatial Reasoning Framework*. In: *COSIT 2011*, *Lecture Notes in Computer Science* 6899, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 210–230, doi:10.1007/978-3-642-23196-4_12.

[5] Eliseo Clementini, Jayant Sharma & Max J Egenhofer (1994): *Modeling Topological Spatial Relations: Strategies for Query Processing*. *Computers and Graphics* 18(6), pp. 815–822, doi:10.1016/0097-8493(94)90007-8.

[6] Anthony G. Cohn, Brandon Bennett, John Gooday & Nicholas Mark Gotts (1997): *Qualitative Spatial Representation and Reasoning with the Region Connection Calculus*. *GeoInformatica* 3(1), pp. 275–316, doi:10.1023/A:1009712514511.

[7] ESRI (2020): https://desktop.arcgis.com/en/arcmap/latest/manage-data/using-sql-with-gdbs/relational-functions-for-st-geometry.htm.

[8] Willem Robert Van Hage, Jan Wielemaker & Guus Schreiber (2010): *The Space Package: Tight Integration between Space and Semantics*. *Transactions in GIS* 14(2), pp. 131–146, doi:10.1111/j.1467-9671.2010.01187.x.

[9] Jiawei Han, Ling Liu & Zhaohui Xie (1994): *LogicBase: A Deductive Database System Prototype*. In: *CIKM '94: Proceedings of the Third International Conference on Information and Knowledge Management*, pp. 226–233, doi:10.1145/191246.191285.

[10] Theofilos Ioannidis, George Garbis, Kostis Kyzirakos, Konstantina Bereta & Manolis Koubarakis (2019): *Evaluating Geospatial RDF Stores Using the Benchmark Geographica 2*. *arXiv:1906.01933 [cs]*, doi:10.1007/s13740-021-00118-x.

[11] Shafat Khan & Mutahar Syed (2017): *Empirical Evaluation of ArcGIS with Contemporary Open Source Solutions - A Study*. *International Journal of Advance Research in Science and Engineering* 6, pp. 724–736.

[12] Kostis Kyzirakos, Manos Karpathiotakis & Manolis Koubarakis (2012): *Strabon: A Semantic Geospatial DBMS*. In: *The Semantic Web – ISWC 2012*, 7649, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 295–311, doi:10.1007/978-3-642-35176-1_19.

[13] Beidi Li, Mehul Bhatt & Carl Schultz (2019): *λProlog(QS): Functional Spatial Reasoning in Higher Order Logic Programming*. In: *14th International Conference on Spatial Information Theory (COSIT 2019)*, pp. 26:1–26:8, doi:10.4230/LIPIcs.COSIT.2019.26.

[14] Werner Nutt (1999): *On the Translation of Qualitative Spatial Reasoning Problems into Modal Logics*. In G. Goos, J. Hartmanis, J. van Leeuwen, Wolfram Burgard, Armin B. Cremers & Thomas Cristaller, editors: *KI-99: Advances in Artificial Intelligence*, *Lecture Notes in Computer Science* 1701, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 113–124, doi:10.1007/3-540-48238-5_9.

[15] OpenStreetMap contributors (2017): *Planet Dump Retrieved from https://planet.osm.org*.

[16] Gilles Pesant & Michel Boyer (1999): *Reasoning about Solids Using Constraint Logic Programming*. *Journal of Automated Reasoning* 22, pp. 241–262, doi:10.1023/A:1006080931326.

[17] Paul Ramsey & Mark Leslie (2012): https://postgis.net/workshops/postgis-intro/de9im.html.

[18] Markus Stocker & Evren Sirin (2009): *Pelletspatial: A Hybrid RCC-8 and RDF/OWL Reasoning and Query Engine*. In: *In: OWLED 2009*, Springer.

[19] Przemysław Andrzej Wałęga, Carl Schultz & Mehul Bhatt (2017): *Non-Monotonic Spatial Reasoning with Answer Set Programming Modulo Theories*. *Theory and Practice of Logic Programming* 17(2), pp. 205–225, doi:10.1017/S1471068416000193.