GeomAttribute

A QGIS Attribute Table with a Geometry Column

Philip Whitten

Jan 26, 2019

GeomAttribute: A QGIS Attribute Table with a Geometry Column

An Attribute Table with a geometry describing column that illuminates those geometries with *empty* or *null* values for the active vector layer in a QGIS workspace.

The plugin's development repository is on Github: https://github.com/PhilipWhitten/geomAttribute

The plugin can be downloaded from the QGIS repository: http://plugins.qgis.org/plugins/geomAttribute

The help documentation is hosted online: http://philipwhitten.github.io/geomAttribute

The initial version of the plugin was created in 2018 by Philip Whitten as part of a Penn State MGIS Capstone Project supervised by James O'Brien.

Python scripts contained within this documentation were tested using the Sphinx doctest extension (https://www.sphinx-doc.org/en/master/usage/extensions/doctest.html) using a standard network installation of QGIS version 3.2.2 in a Microsoft Windows environment.

CONTENTS:

CHAPTER

ONE

PREFACE

I am in the closing stages of a MGIS at Penn State and needed a topic for a capstone project. What sort of topic? I am gainfully employed as a GIS Officer at a local government bureau, but, my employer was not willing to offer a topic nor were they willing to offer open access to curious datasets. I have a passion for maps, cartography and spatial analysis, but, previous attempts to combine my passion with requirements for academia often ended in a feeling of futility. From my employment, I realised that my colleagues and I occasionally loose hours because some of our datasets were **corrupted** by *null* or *empty* geometries. I also realised that I was blissfully ignorant of the concept of *null* and *empty* geometries as they were not included in my GIS education. So, I wanted to learn a bit about *null* and *empty* geometries in the hope that I can be more capable in my employment and so that I can feed a curiosity.

The concept of a QGIS plugin that exposed *null* and *empty* shapes evolved quickly. A plugin is tangible and deliverable as opposed to curiosity which is instinct and always expanding. QGIS is the primary desktop GIS software where I work and I was keen to learn how to automate and develop QGIS for specific tasks. In my GIS degree we had used Python to automate tasks in ArcMap, and, C# to make apps for ArcMap, but, we had not dabbled in QGIS. So, this project was also an adventure in QGIS scripting.

What comes next? From this project I have learnt that much of the GIS software and the GIS community is GIS centric. After all, to create a new record in a dataset with either QGIS or ArcMap one creates the shape of a feature, and, *subsequently* populates the other attributes. To work in reverse and create the shape of a feature as the last step in a workflow is not only unconventional, but, also nontrivial. But here's the point, whenever I have to create geospatial records I am always given a list of attributes and the shape is the last attribute to be created, the other attributes are already known and are re-entered. Furthermore, most of the major desktop GIS packages report set operations between datasets differently to established database conventions. I would like to work towards making QGIS and other software packages less GIS centric so that geospatial datasets become more ubiquitous and more open to other professional communities! To become less GIS centric you need to embrace the *null* and exploit the *empty*.

CHAPTER

TWO

INTRODUCTION

A dataset is a collection of records, where each record has a defined number of elements and the data type of each element is defined. The elements are commonly referred to as attributes. For example, the dataset of street names shown in Table 2.1 consists of records that have the attribute headings of *"Name"* and *"Street Type"* with the data types *letters and spaces* and *letters* respectively. In this example, *Street Type* contains values from the set *{'Lane', 'Road', 'Street', ''}*. The first three values *'Lane'*, *'Road'* and *'Street'* are obvious – but what about the last value of *''*? **''** *is a deliberate empty set of character values!* In our world there are many streets that have no type, and, indeed there are many that have no name. Here, the use of an *empty* set of characters, or *''*, indicates that the *Street Type* attribute has no type, not that the type is unknown (*null*).

Table 2.1: An example of a road name dataset.

Name (letters and spaces)	Street Type (letters)
Picton	Road
Menangle	Street
The Boulevard	[1]

Geospatial datasets contain one or more values that refer to a location on earth. For the majority of geospatial datasets, the location consists of one or more points, lines, and/or polygons, that are referenced to a coordinate system that is a projection of the earth's surface.

For this project, any dataset element type that stores the geospatial shape with respect to a referenced coordinate system is called a **geometry**. Any *empty* geometry element is a geometry that doesn't have any of the vertices which are required to construct a shape.

In many organizations geospatial datasets are contained within enterprise databases where frequently the same brand of enterprise database is used elsewhere within the same organization to contain non-spatial datasets. For example, a local government office may use one or more MicroSoft SQL Server installations as a dataset repository for: a content management system; a customer relationship management system; a land management system; an asset management system; and, a Geographic Information System (GIS).

QGIS [3] is a computer program that among other things is used to view, create and edit the geometry values within geospatial datasets. For many data sources QGIS does not: Parse *null* and *empty* geometry values equivalently for different data storage formats; does not directly show which records within a

[1] The street name *Boulevard* is also a type of street, consequently the *Street Type* field is *empty*.

dataset have *null* or *empty* geometry elements; and, does not always process *null* and *empty* geometry elements as specified by international or open geospatial standards.

A conceptual description of datasets, geometry data types, and, *null* and *empty* data values are outlined in the *Background* section.

CHAPTER THREE

BACKGROUND

The concepts of relational datasets in general, and, of geospatial data values and data types are introduced. An emphasis is placed on *empty* and *null* geospatial data values. QGIS software is also introduced. Specific QGIS problems that are associated with geospatial data values and data types are kept within the *QGIS Data Provider Data Parsing Problems* section.

3.1 Relational Datasets

A *set* is a collection of distinct objects. For example, a box of apples is a *set* of apples, and, the *set* of countries in North America contains: Canada; United States of America; and, Mexico. By convention, sets are symbolized by enclosing within curly brackets. Hence:

$$North\ American\ Countries = \{Canada, United\ States\ of\ America, Mexico\}$$

A dataset is any set where each element is restricted to one data type *and* where each element belongs to the same universal set.

A relational dataset is a collection of sets where:

1. The number of objects (elements) in each set is the same; and,
2. A one to one relationship exists between elements of different sets.

An example relational dataset showing the Country Name, Country Abbreviation and Country Population for the countries in North America is shown in Table 3.1. This relational dataset comprises of the three sets: country names; country name abbreviations; and, country populations. A one to one relationship exists between the elements for each of these three sets. Hence, the country with the name of *Canada* has a one to one relationship with the country abbreviation *CAN* and the country population *3563000*. The combination of the relationship and the set values for a specific country is represented as a row in a table that is referred to as a *record*.

Table 3.1: Example of a North America Countries dataset.

Country Name (letters and spaces)	Abbreviation (3 upper case letters)	Population (integer)
Canada	CAN	3563000
United States of America	USA	32663000
Mexico	MEX	12458000

Essential to any set is a definition or description of what type of objects can be a member. For example, an apple which is a valid type of the set of fruit can't be a member of a set of countries. For any dataset, both the data type and additional constraints are often used together to define the universal set. For example, the data type for *Country Name* in Table 3.1 is any combination of letters and spaces, whilst the data type for *Abbreviation* is any combination of 3 upper case letters.

Although a *relational dataset* consists of *multiple sets* of data where the elements of each set are related, it is ubiquitously referred to as a **dataset**.

3.1.1 Geospatial Dataset

A geospatial dataset refers to any dataset where one or more of the composite sets refer to a location on the earth's surface. This project scope is restricted to those geospatial datasets where the location on the earth's surface is represented by one or more points, lines or polygons that are located by vertices (co-ordinates) and stored as vectors. These points, lines and polygons are collectively referred to as shapes or *geometries*[6]. Datasets that include one or more sets of geospatial vectors are referred to as *Vector Datasets* by the GIS community. A vector geospatial dataset is a subtype of geospatial dataset where the geospatial sets can be located graphically on the earths surface by the application of coordinates.

3.2 Geometry Data Types

All datasets contain some restriction on the type of data each constituent set may contain. From a software perspective, a restriction of type is essential for minimizing both the storage size of the dataset and the response time for a dataset query. Analagous to specific data types for storage of numbers, text or dates there are data types specifically used for the storage of geospatial geometries [5]. Similarly, just as there can be specific data types for numbers including signed and unsigned integers, float, and, decimal numbers, there are also specific data types for different types of geometries, with the type often referring to how the geometry is constructed.

For any data source the geometry data types that are availabe for use can be shown schematically as a hierarchy like the one shown in Fig. 3.1. Within this hierarchy, the possible data types are described by the labels in the boxes. Essential to all such hierarchy's, a set of data of a declared type may consist of any type below it on the hierarchy. Hence for the hierarchy shown, if a set of data has a declared type of *Geometry Collection* then any data element within it may consist of *Geometry Collection*, *Multi-Point*, *Multi-Curve*, *Multi-Line*, *Multi-Surface*, and, *Multi-Polygon*. Similarly, if a set of data has a declared type of *Point* than it may not contain a *Polygon* nor a *Line* as neither are below the sub-type *Point* on the hierarchy.

The single part constrained geometry subtypes in the lower part of Fig. 3.1 are referred to as *Primitive Types* and must contain only one single part geometry per set element. In contrast, the *Multiple Parts* geometries may consist of one *or* many parts per feature. For example, a feature of the *"Multi-Point"* geometry sub type may have one point, no points or multiple points. Another characteristic of the single part primitive types is that the *Line* and *Polygon* subtypes may only exist of straight line segments between coordinates.

In reality there may be many more geometry subtypes than the simplified hierarchy shown in Fig. 3.1. For example, some common additional subtypes for datasets are created for sets of geometries that incorporate elevation, or, for lines that are constructed from curves as opposed to straight line segments. In another variation, some geometry hierarchy's including Microsoft SQL server and QGIS don't isolate primitive geometry sub-types from non-primitive sub-types.

Many GIS data source standards, and, many GIS software have a geometry subtype hierarchy that is **similar** with Fig. 3.1, including the inheritance diagram for QGIS's QgsAbstractGeometry[2] [12].

[6] For SQL relational databases, the term geometry is restricted to those shapes that are located by cartesian coordinates.

[2] https://qgis.org/api/classQgsAbstractGeometry.html

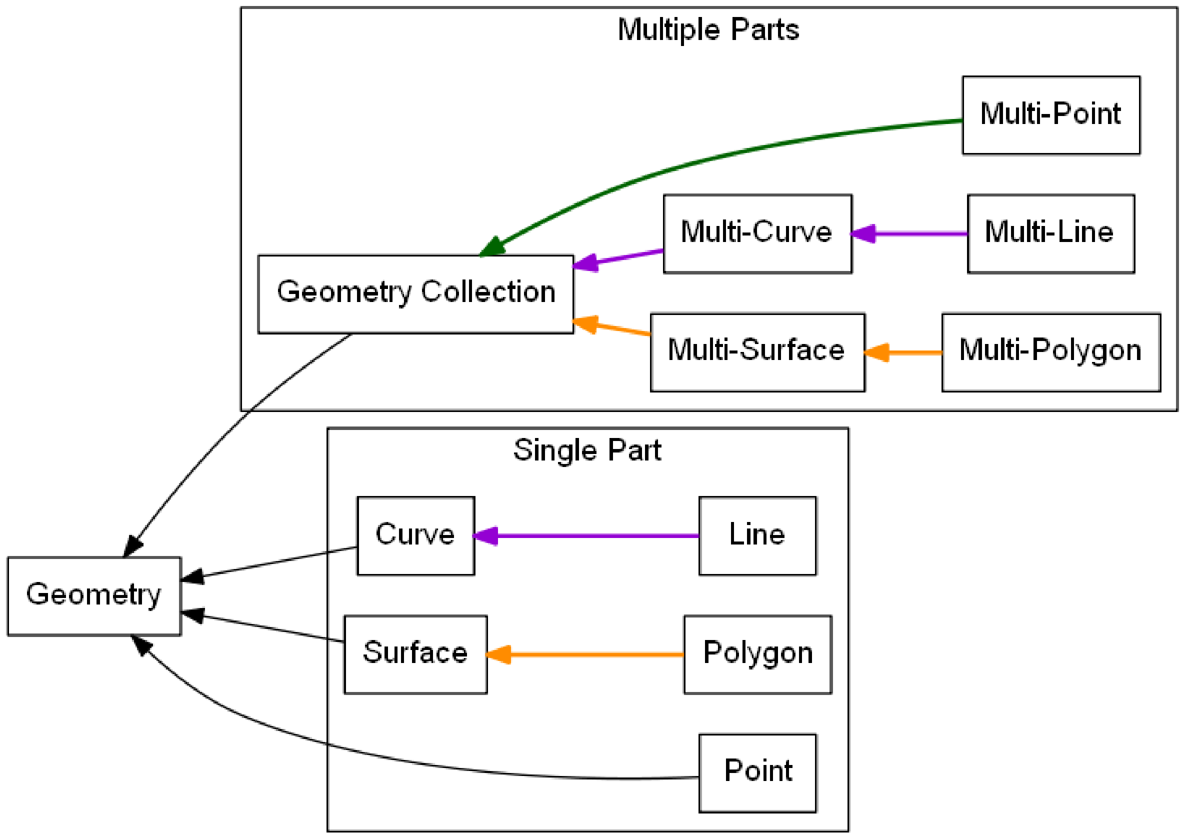


Fig. 3.1: GIS Geometry subtype hierarchy. Adapted from [5]. The more conventional term *"LineString"* that is used in the QGIS API and [5] is replaced here with *"Line"* for clarity.

3.3 Geometry Data Values

For any data type there exists a universal set of valid values. For example, a set of birthday dates must be restricted to valid dates. Hence, a birthday on the 30th of February is not valid as the 30th of February is not part of the universal set of dates. Similarly, a valid geometry should be located within the boundaries of the coordinate system that it is referenced to. **Empty** and **null** are two values that may be part of many sets of data and for geometry data values could be fairly described as being:

1. controversial;
2. miss-understood; and,
3. best avoided.

3.3.1 Empty

A box of apples can be described as a set of apples. An *empty* Apple box represents an *empty* set of apples. An *empty* geometry element is a geometry that has no coordinates. Whether an *empty* element is a valid member of a set depends on the context. For example, if a study of chickens hatching from eggs recorded the date that each chicken hatches for a set of 10 eggs, than the hatch date element for each egg is *empty* before the chicken hatches. It is *known* that the chicken has not hatched.

All *empty* set values including an *empty* geometry data value are place holders for when it is *known* that an element does not exist [21]. For example, consider the intersection [10] of the *Blue Crosses* and the *Red Circles* with the two squares shown in Fig. 3.2. Both of the *Blue Crosses B1* and *B2* intersect the *Left square*, and, the *Blue Cross B3* intersects the *Right square*. The intersections of the *Squares* and *Blue Crosses*, and the *Squares* with *Red Circles* are summarized by the datasets shown in Table 3.2 and Table 3.3. As shown in Table 3.2 the *Left Square* intersects with the *Blue Crosses B1* and *B2* as represented by the subset *{B1, B2}*. Similarly, it is reported in Table 3.3 that the *Left square* intersects the subset of *Red Circles {R1}*. In contrast, also in Table 3.3 it is shown that the *Left square* does not intersect with any *Red Circles* as shown by the *empty* set *{ }*. Here the *empty* set *{ }* shows that it is known that no intersection occurs. The reporting of those combinations where intersections are known to not occur as shown in Table 3.3 follows the convention used by most SQL type relational databases for all set intersections regardless of whether they are geospatial or not. In contrast, the convention for many GIS desktop software including QGIS and ArcGIS is to only show those combinations where intersections are known to occur (are *True*). Table 3.4 follows the geospatial intersection convention of QGIS and shows only those combinations in Table 3.3 that do intersect. Analyzing those sets that don't intersect (*empty* sets) can be insightful for data workflow problem solving .

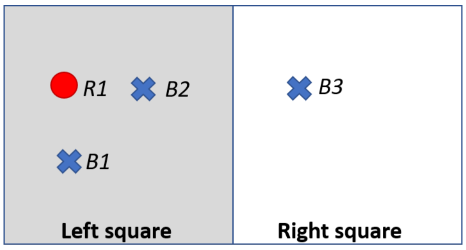


Fig. 3.2: The location of *Blue Crosses* and *Red Circles* in the "Left Square" and the "Right Square".

Table 3.2: The intersection of the *Squares* and the *Blue Crosses*.

Square	Blue Crosses
Left square	{B1, B2}
Right square	{B3}

Table 3.3: The intersection of the *Squares* and the *Red Circles*.

Square	Red Circles
Left square	{R1}
Right square	{ }

Table 3.4: The intersection of the *Squares* and the *Red Circles* where the intersection is *True*.

Square	Red Circles
Left square	{R1}

The real utility of *empty* geometry values is realised when the intersection of all the squares and both types of points (*Red Circles* and *Blue Crosses*) are collated in one dataset as shown in Table 3.5 as opposed to Table 3.6. By using the *empty* set *{ }* as a place holder for the known non-intersection of *Red Circles* with the *Right square* the sets of *Blue Crosses* and *Red Circles* are maintained as separate columns in Table 3.5. Although this approach is efficient and intuitive it is not suitable when there are a large number of point types as the number of columns has a linear relationship to the number of point types.

Table 3.5: The intersection of each point type set and the set of squares. Note that the point type sets are maintained as separate sets (columns).

Square	Point type	
	Blue Crosses	Red Circles
Left square	{B1, B2}	{R1}
Right square	{B3}	{ }

Table 3.6: The intersection of all point types and the squares with one combined set of point types, and, where the intersection is *True*.

Square	Point type	Geometry
Left square	Blue Crosses	{B1, B2}
Right square	Blue Crosses	{B3}
Left square	Red Circles	{R1}

Datasets With Multiple Geometry Sets

Much of the GIS community work with the restriction of a single geometry set per dataset (a single geometry column within a table). It is difficult to have multiple geometry attributes without also allowing *empty* geometry values. Next, I will examine the advantages and disadvantages of multiple geometry attributes.

The fundamental advantage of multiple geometry attributes is they facilitate topology. Topology refers to how the constituent parts of a system are interrelated or arranged. The location of points within squares shown schematically in Fig. 3.2 is an example of topology as it shows how the points are related to the squares. Table 3.5 shows the topological association of point type by square type, but, uses two geometry columns to do so. Table 3.6 shows all of the True intersections shown in Table 3.5 using only one geometry column. Examination of Table 3.6 reveals that the relationship between a type of point (e.g. Blue Crosses) and the Square type (e.g. Left Square or Right Square) has to be reported as two separate relationships (two separate records) and a user is left with the task of mentally associating these two relationships. Clearly, without using multiple geometry columns establishing topological relationships is less intuitive.

A major disadvantage of multiple geometry columns is that they are not supported by many GIS software or GIS data sources. For example, ArcGIS does not support multiple geometry columns in any capacity, QGIS treats each geometry column as an unrelated dataset, and, the ubiquitous shapefile can only contain one geometry column. So, by adopting multiple geometry columns a user potentially isolates themself from a significant portion of the GIS community.

Set Operations in Microsoft SQL or PostGIS Produce Empty Geometry Values

There are several fundamental set operations that are used to construct new sets from existing sets regardless to what it is a set of [17]. Within a database, it is ideal if a set operation is consistent for all data types, regardless of whether they are geospatial or not. As introduced in the section *Empty* some databases generate *empty* geometry values for intersection set operations. Other universal set operations include Union and Except. Microsoft SQL Server and PostGis are examples of databases that can generate *empty* geometry values for set operations. Both of these databases were developed from SQL conventions and their generation of *empty* values for geometries is consistent with set operations for other data types.

Consider the intersection of the points *Pt1* and *Pt2* with the circle shown in Fig. 3.3 which is an example of an intersection between two geometry datasets. ‘Pt1’ lies within the *circle*, and, hence intersects the *circle*. *Pt2* lies outside the *circle* and does not intersect it. In both QGIS v3.x and ArcGIS desktop, the intersection of a points dataset containing *{Pt1, Pt2}* and a dataset containing the *circle* will return only those records that intersect, hence, only *Pt1* is returned. In contrast, with Microsoft SQL Server, the intersection returns both the records that do and don’t intersect with a dataset containing both *Pt1* and *Pt2*.

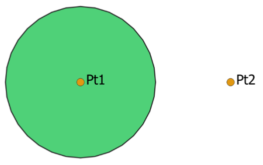


Fig. 3.3: A schematic of the intersection of points *Pt1* and *Pt2* with the *circle*.

There is no error with either of the different approaches to intersection taken by QGIS 3.x and Microsoft SQL Server. QGIS 3.x includes the selection logic step that removes those records that don’t intersect. For *Pt2*, Microsoft SQL server is returning an *empty* geometry confirming that no intersection exists, as shown by the following Transact-SQL script[5]:

```
DECLARE @circle geometry = 'CURVEPOLYGON (CIRCULARSTRING (0 1, 1 2, 2 1, 1␣
↪0, 0 1))';
DECLARE @Pt2 geometry = 'POINT (3 1)';
SELECT @Pt2.STIntersection(@circle).ToString();
GEOMETRYCOLLECTION EMPTY
```

This intersection example shows the ease with which one can inadvertently generate *empty* geometry values in Microsoft SQL Server, and, these *empty* geometry values will be parsed to QGIS.

[5] In both Microsoft SQL server and PostGis the geometry type that is *empty* is recorded [20][22]). Sometimes the geometry type gets changed to the generic ‘GEOMETRYCOLLECTION’ by set operations.

3.3.2 null

null is the most common value (element) recorded by many disciplines and data sources for *unknown* data values[7]. For example, if a study of chickens hatching from eggs recorded the hatch date of each chicken hatching for a set of eggs, then for a specific egg the hatch date value is *null* (unknown) if the hatch date was not recorded, but, the chicken has hatched. Strictly speaking a *null* hatch date can be any value from the universal set of hatch dates including *Empty* allowing for eggs that never hatched.

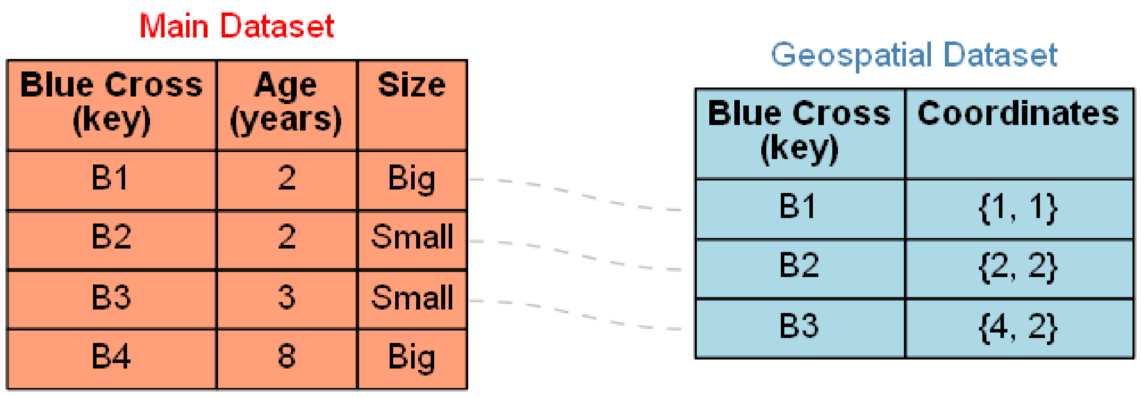
The most useful feature of *null* values is that they enable incomplete datasets. For example, consider the *Blue Crosses* dataset shown in Table 3.7 where the coordinates for *B4* are unknown. Datasets like Table 3.7 can stem from requests to georeference existing datasets where the georeferencing is incomplete.

Table 3.7: The age, size and coordinates for Blue Crosses.

Blue Cross	Age (years)	Size	Coordinates
B1	2	Big	{1, 1}
B2	2	Small	{2, 2}
B3	3	Small	{4, 2}
B4	8	Big	*null*

Using Joins to Eliminate null

Many GIS datasets do not allow *null* geometries. Having a dataset constraint that prevents *null* geometries does not imply that all geometries are known, it only means that the dataset can't have a *null* geometry data value. The prevention of *null* geometries without knowing all of the geometries is achieved by using multiple datasets that include a geometry only dataset that has a relationship with a non-geometry dataset as shown in Fig. 3.4. The relationship is typically achieved by the use of a unique value *key* that is used in all related datasets to distinguish each relationship across the datasets. *Joins* refer to the process of forming a new dataset from multiple datasets by the use of a relationship. The dataset shown in Table 3.7 can be created from the datasets shown in Fig. 3.4 by application of an *outer join*.


Main Dataset

Blue Cross (key)	Age (years)	Size
B1	2	Big
B2	2	Small
B3	3	Small
B4	8	Big

Geospatial Dataset

Blue Cross (key)	Coordinates
B1	{1, 1}
B2	{2, 2}
B3	{4, 2}

Fig. 3.4: Table 3.7 presented as two separate datasets where *null Coordinates* data values are not permitted in the geospatial dataset. The *Blue Cross* keys are used to register relationships symbolised by the grey dashed lines between specific records in the two datasets.

The use of a geospatial specific dataset with a key like that shown in Fig. 3.4 enables geometry to be a variable. For example, take the Blue Cross *B1*. This Blue Cross may represent a boat at sea. Hence, at different points in time, *B1* may have different coordinates (Table 3.8). Most geospatial datasets have

[7] Python uses *None* instead of *null*, but, PyQT uses *NULL* as a QVariant, so, PyQGIS scripts may have a mixture of *None* and *NULL* depending on the origin of the classes in use.

geometries that are variable as our technology for recording and referencing location is improving. For example, an allotment of land may be static as it is relative to reference points, but, the mapped location and hence the recorded geometry for the allotment of land will change as the location of the reference points is refined to a higher accuracy. Whether a dataset owner should track changes to a geometry is a dataset specific question.

Table 3.8: The coordinates of the Blue Crosses for yesterday and today.

Blue Cross	Coordinates	
	Yesterday	Today
B1	{1, 1}	{2, 2}
B2	{2, 2}	{1, 2}
B3	{4, 2}	{4, 1}

Reasons For Preventing null Geometries

The reasons for preventing *null* geometries include: GIS feature creation is simpler without *null* geometries; *null* geometries can't be mapped; and, Boolean logic can't be applied to *null* values.

For QGIS and many desktop GIS systems, records are created by drawing the geometry and subsequently entering the record's other data values. This geospatial geometry centered approach intuitively keeps the related computer programming simple in comparison to any approach that allows a user to enter any of the data values including the geometry in the sequence that they choose.

There is no accepted universal approach to mapping a *null* geometry. It is obvious that if a geometry is *null* then using a defined symbol at a single location is a miss-representation. There is active research into approaches for mapping the unknown [23].

Whether a dataset permits *null* values directly affects the type of logic applied to the dataset for set operations. *Boolean* logic, also referred to as two value logic, allows only for *True* or *False* answers to set operations. *Boolean* logic can't be applied when the answer is unknown. When *null* values are permitted, *Three Value* logic is required for set operations. The intersection of the squares with the two subsets of *Blue Crosses {B1, B2, B3}* and *{B1, B2, B3, B4}* that are described in Table 3.7 is shown in Table 3.9. For *{B1, B2}* it is *True* that they intersect the *Left square*, whilst it is also *True* that *B3* does not intersect the same square, however, it is *null* (unknown) whether *B4* intersects the *Left Square*. Compounding the implementation of *Three Value* logic is the fact that different database sources have different implementations of it differently leading to it's widespread avoidance. The different implementations of *Three Value* is not specific to geometry data values and is a problem for all data value types including generic types like integer or date [19][25]. In summary, even when *null* values are permitted in datasets, the records associated with them are typically excluded from set operations.

Table 3.9: The intersection of the squares with the set *{B1, B2, B3, B4}* as shown in Table 3.7 using different types of logic.

Logic	Test	Square	
		Left	Right
Boolean	True	{B1, B2}	{B3}
	False	{B3}	{B1, B2}
Three Value	True	{B1, B2}	{B3}
	False	{B3}	{B1, B2}
	Unknown	{B4}	{B4}

Reasons For Allowing null Geometries

Although the majority of GIS systems do not promote the use of *null* geometry data values, there are several reasons supporting their adoption: they assist the adoption of geospatial datasets by enterprise databases; they illuminate the unknown; they facilitate a simpler dataset structure; and, many GIS systems allow *null* for other sets.

Facilitating the adoption of geospatial datasets by enterprise databases is a compelling reason to employ *null* geometry values. For example, consider a commercial database used for land rates (also known as land taxes) at a local government office. The current approach of many rates databases is to mimic Fig. 3.4 where the tables for the land registry are maintained within a non-spatial land registry database, and, the tables for the georeferenced land allotments are in a GIS database with the databases being joined. This approach facilitates bureaucracies where separate teams of people maintain each database, and, where the land registry database does not adopt spatial functionality as the data is located in a separate database. Whilst using a separate GIS database allows the local government office freedom to acquire practically any common GIS system, it comes at a cost of minimal inbuilt spatial capability in the non-spatial land registry software.

Allowing *null* has the potential to expose the unknown. By exposing the unknown, it it is evident where further data capture is required, and, it is more likely that any spatial analysis will also establish the degree of dataset completeness.

A disadvantage of the lookup table approach is the required maintenance of lookup keys and database schema that are essential for performing database queries that include spatial and non-spatial attributes. Without lookup tables the database structure is simpler leading to fewer errors and fewer joins when constructing queries.

Many GIS systems allow *null* values for non-geometry sets (for example *null* values for age in an asset dataset), but, disallow them for a geometry set within the same dataset. Excluding *null* from sets that are not unique keys on a piecemeal basis can cause confusion. The rational for using *null* values should be consistent for all non-key sets within a dataset, and, within a database.

3.4 QGIS

QGIS is self-promoted as an “Open Source Geographic Information System” [3]. QGIS is used for creating, manipulating and publishing spatial data sets by many organisations. QGIS was created by Gary Sherman in 2002 [16][27]. In 2007 it became a project of the Open Source Geospatial Foundation with version 1 being released in January 2009 [16]. The version of QGIS used in this project, Version 3, was released in February 2018 [16]. Version 2 of QGIS employed Python 2 for scripting and PyQT4 for the Graphical User Interface (GUI). Version 3 of QGIS employs Python 3 for scripting and PyQT5 for the GUI. QGIS version 3 is self-described as a “huge overhaul and cleanup” of QGIS version 2 [9]. Many of the Python scripts configured for version 2 of QGIS no longer work with version 3 with much of the legacy sub-version support dropped.

A feature that distinguishes QGIS from many desktop GIS platforms including MapInfo and ArcMap is that QGIS does not incorporate or associate with a custom data source. For example, MapInfo is associated with TAB files and ArcMap is associated with file and personal geodatabases. In contrast to MapInfo or ArcMap, QGIS has a data source provider philosophy which it uses to provide a consistent user interface for a broad range of data sources.

3.4.1 QGIS Data Providers

A data provider is a software library that reads, writes, executes commands and queries one or more data sources. These data sources may be as simple as a text document or spreadsheet, or, complex like a

SpatiaLite or PostGis database. Analysis software packages may have many data providers so that they can access a wide range of data sources.

Each data source has different combinations of data values and data types, and, also different commands that can be executed. Some data providers service several different data sources, whilst, others service a single data source. For GIS, data providers need to negotiate different collections of geometry sub-types, and, different approaches to *null* and *empty* data values. It is the QGIS data provider's task to:

- provide algorithms to parse data values between external data source sub-types and QGIS sub-types; and,
- for some data sources the association of geometry sub-types and it's parsing is per record, not per dataset.

For example, an ESRI Shapefile has only 4 instantiable geometry sub-types and does not permit *empty* geometry values within it's specification (Fig. 3.5). In contrast, a Microsoft SQL server spatial database has 10 instantiable geometry sub-types and permits both *null* and *empty* geometry values (Fig. 3.6). A software package that has providers for both ESRI's Shapefile and Microsoft SQL server spatial database needs to provide a common user experience that is independent of the provider being used as much as feasible.

A software user may need to be aware of the differing complexity of different data sources and the provider's which interact with them if the software engaging the provider does not provide warnings or messages when data types and values are changed *in situ* to provide a smooth and consistent user experience. Note that for reading data sources, any in-situ data changes are only for presentation or analysis and are not written to the data source.

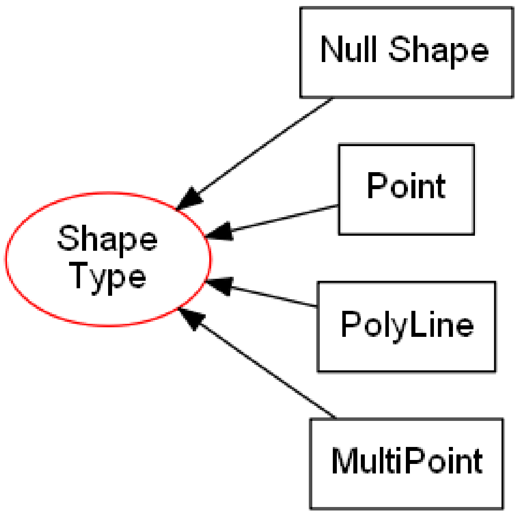


Fig. 3.5: The hierarchy of geometry types for a ESRI shapefile excluding those that include elevation or measurements (adapted from [18]). The *Shape Type* geometry type can't be instantiated directly and is included for consistency.

QGIS parses data to and from many data sources. The algorithms for this data parsing are contained within 18 different data providers. The data providers are scripted within C++ and are excluded from the QGIS Application Programming Interface (API) [14]. To function, the relevant data provider needs to read and write each data type and the appropriate values for each set (column) included in the dataset. The intent of this data parsing is that a user may read, write and analyse data stored in different sources with a common QGIS user interface.

[4] In many SQL databases there are duplicate hierarchy's with separate geometry data types for cartesian and geographic coordinates.

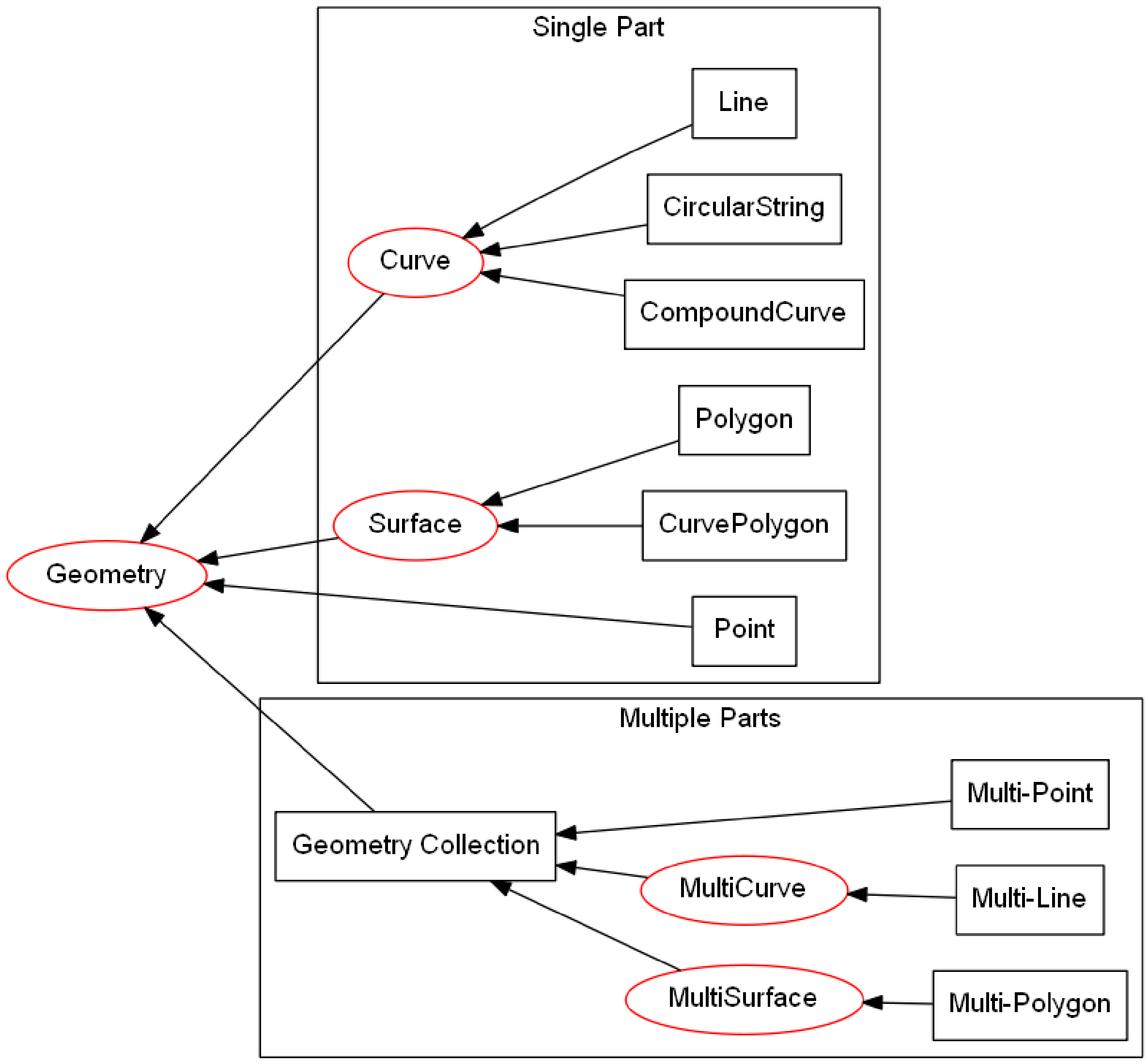


Fig. 3.6: The hierarchy of geometry types for Microsoft SQL server excluding those that contain elevation or measurements (adapted from [7])[4]. Types that can be instantiated are in black rectangles. Types that can't be instantiated are in red ellipses.

Qgis's data provider approach allows QGIS to edit, create or analyse geospatial datasets within enterprise databases in their native format. For example, SMEC Pavement Management Software[3] can be configured to use a Microsoft SQL server database to contain it's pavement datasets. By way of a data provider, QGIS can browse and edit the geometries contained within a SMEC pavement dataset without importing or exporting any dataset, and, without creating additional tables in any database.

In comparison to QGIS's approach of editing the data in it's native format, ESRI's ArcMap requires a user to import a dataset into a geodatabase prior to editing, even though the enterprise database and the geodatabase may be using the same database server [15]. ESRI's approach often leads to lookup tables being used for geospatial data that are joined to non-spatial datasets.

3.5 GIS Professional Awareness

Many GIS professionals are educated and work within GIS dataset schema where both *null* and *empty* geometries are excluded. It is plausible that GIS professionals that are not familiar with *null* and *empty* shapes are ignorant of them when using a GIS where they are permitted. This is a situation that confronted the proponent of this project. Examples of miss-understanding of *null* and *empty* geometry values are widespread on the world wide web. For example, although the coordinates of a *null* geometry are unknown, an article published by ESRI incorrectly states that *an empty geometry exists for any geometry where the coordinates are unknown* [2].

Micosoft SQL server and PostGis both permit *null* and *empty* geometry values unless explicitly excluded by constraints or third party software. Even when a primary dataset contains no *null* or *empty* geometry values, processing of that dataset may produce *null* or *empty* geometry values. Performing set operations like intersections using database methods, and, editing geometry of specific records within QGIS are two ways *empty* geometry values can be created.

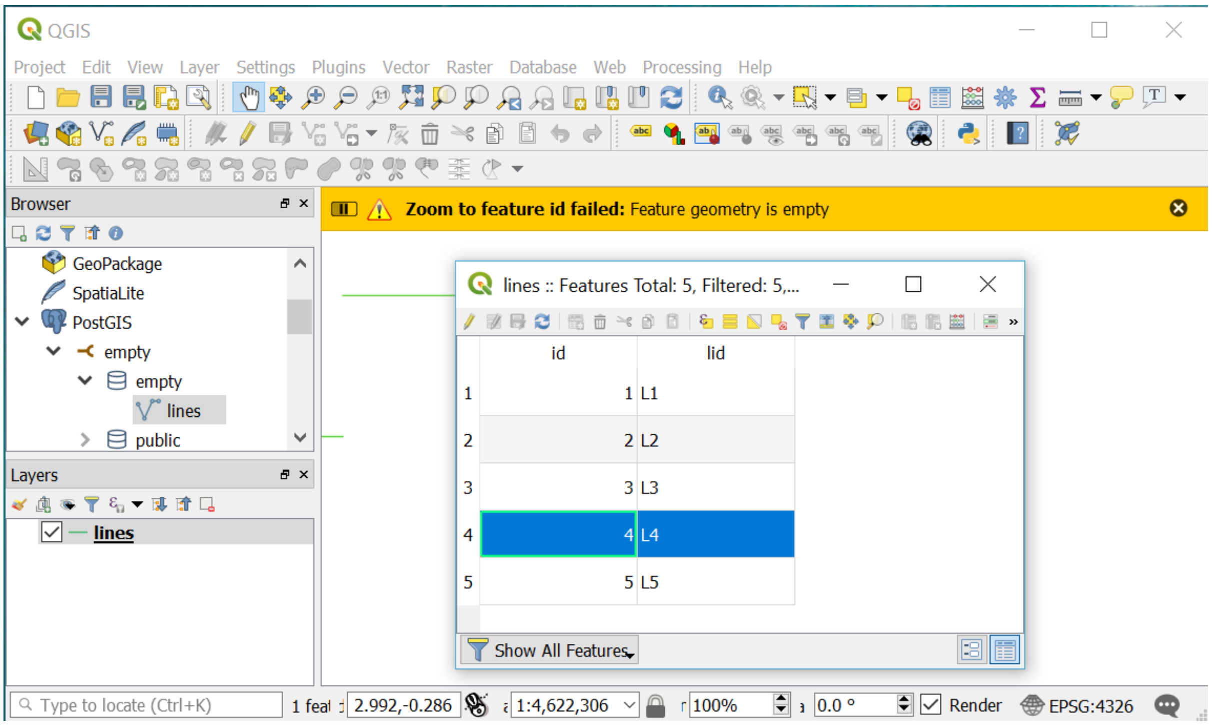
3.5.1 Exposing null or Empty Geometry Values in QGIS

Within QGIS, without using custom expressions or scripts there are only limited scenarios where a user will be alerted to *null* or *empty* geometry values when they are present. Two tools within QGIS where you may find descriptive information of each record are the *"Attribute Table"* and the *"DB Manager"* plugin.

Without using expressions and creating new attributes, the "Attribute Table" does not contain any information about a record's geometry. The only component of the "Attribute Table" that exposes *empty* or *null* geometries is when a user selects *"Zoom to Feature"* for a specific record that has a *null* or empty geometry. In QGIS version 3.x, a warning message is shown on the map canvas when a user attempts to zoom to a feature with an *empty* or *null* geometry (Fig. 3.7).

The DB Manager in QGIS 3.x is a core plugin (it can't be uninstalled). The DB Manager plugin provides database specific information for data sets from a limited number of database sources that includes PostGIS, but, excludes Microsoft SQL Server. The *"Table"* view within DB Manager shows all of the columns within the table including the geometry attribute. Within the geometry column it gives the geometry type and exposes *null* geometries as *"NULL"* (Fig. 3.8). In contrast, *empty* geometries are referred to by their geometry type. Note that the DB Manager's approach is theoretically correct as *null* geometries have an unknown geomtry type, whilst *empty* geometries have a known geometry type.

[3] http://www.smec.com/en_au/what-we-do/sectors/transport/pavement-management-systems



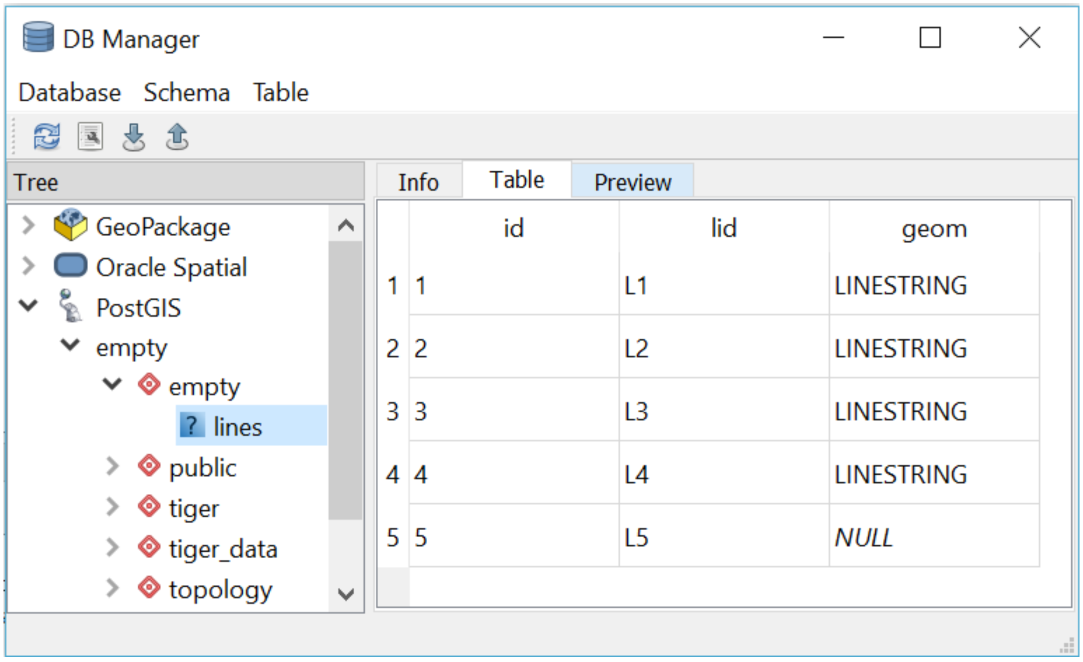
Zoom to feature id failed: Feature geometry is empty

lines :: Features Total: 5, Filtered: 5,...

	id	lid
1	1	L1
2	2	L2
3	3	L3
4	4	L4
5	5	L5

1 feat 2.992,-0.286 1:4,622,306 100% 0.0 °

Fig. 3.7: A warning message is displayed in QGIS when a user attempts to zoom to a record with an *empty* geometry.



	id	lid	geom
1	1	L1	LINESTRING
2	2	L2	LINESTRING
3	3	L3	LINESTRING
4	4	L4	LINESTRING
5	5	L5	*NULL*

Fig. 3.8: The output from the DB Manager plugin in QGIS 3.x for a dataset that contains both *empty* and *null* geometry values.

CHAPTER

FOUR

QGIS DATA PROVIDER DATA PARSING PROBLEMS

The parsing of datasets by QGIS data providers is silent and can change both geometry sub-type and the geometry data values of *null* and *Empty* (Table 4.1).

Changes of geometry sub-type may be required because the data source doesn't support the mixing of specific sub-types, or, because the source sub-type may not be supported at all. For example, in Table 4.1, neither Geopackage nor Spatialite permit the mixing of *Line* and *Multi-Line* subtypes, so, the data provider defaults to the *Multi-Line* subtype for the creation of a new dataset, and, all records with a *Line* subtype are changed to a *Multi-Line* subtype. In an another example, Shapefile does not support the *Line* subtype at all, with all records with a *Line* subtype changed to a *PolyLine*[9].

For geometry data vales of *null* and *empty*, values are changed as the data source does not permit both types, or, because the data provider contains errors. In Table 4.1 it is shown that with the exception of the PostGis format, either the parsing of the known geometry data value of *empty* is changed to the unknown value of *null*, or, visa versa. The replacement of known with unknown, or unknown with known can cause erroneous analysis and interpretation. Without experience errors may be introduced into datasets by the parsing of data by QGIS's data providers.

Table 4.1: Appending of non-empty single part multi-line, non-empty line, empty line and *null* geometry records by QGIS to 5 popular data sources.

QGIS memory	Geopackage	Shapefile	Spatialite[14]	PostGis[12]	MS SQL[12]
Multi-Line	Multi-Line	Polyline	Multi-Line	Multi-Line	Multi-Line
Line	**Multi-Line**	**Polyline**	**Multi-Line**	Line	Line
Empty	*Empty*	**null**	*Empty*	*Empty*	**null**
null	**Empty**	*null*	*Empty*	*null*	*null*

4.1 Parsing Geometry Data Types

The parsing of geometry records by the QGIS data providers often requires changing the geometry sub-type. For example, consider the parsing of geometry subtypes between ESRI's Shapefile, QGIS, and, SpatiaLite (Fig. 4.1). SpatiaLite and Shapefiles have a single geometry sub-type defined for a dataset[13] which is simpler than data sources like Microsoft SQL Server and PostGIS where the geometry sub-type can vary for each record.

[9] A Shapefile's *PolyLine* is essentially the same as a *Multi-Line* for the scope of this report.

[14] SpatialLite table has a *Multi-Line* geometry data type.

[12] The geometry type saved by PostGis and Microsoft SQL server depends on: geometry constraints within the database; the use of a *Geometry columns* lookup table; and, what geometry types already exist within the respective datasets.

[13] Technically in a Shapefile the geometry sub-type is recorded for each record, but, the technical specifications state that "All non-null shapes must be of the same shape type" [18].

The hierarchy for the ubiquitous Shapefile shown in Fig. 3.5 is vastly different to that for QGIS [12]. For constructing a single part line, QGIS has the four geometry subtypes of *"Multi-Line"*, *"Circular-String"*, *"Compound-Curve"*, and, *"Line"*, whereas Shapefile only has the single subtype of *"Poly-Line"*[8]. Hence, in a QGIS editing session, a user may create a line using any of QGIS's four line subtypes, but, the line will only be recorded as a *"PolyLine*" and it is the Provider's task to inform QGIS of this requirement (Fig. 4.1).

In comparison to a Shapefile, SpatiaLite has the *Multi-Line* and *Line* sub-types and QGIS has to distinguish between these. The manner in which QGIS distinguishes between a *Multi-Line* and a *Line* for parsing to SpatiaLite is primitive. A single part QGIS *Multi-Line* can't be parsed to a SpatiaLite *Line* as the provider refers to the geometry type and not the number of parts, but, a "Line" will be converted to a *Multi-Line* by silently changing it's geometry type if it is parsed to a SpatiaLite *Mutli-Line* data source. In a similar manner, *Circular-String* and *Compound-Curve* sub-types will be converted to *Line* sub-types, and then *Multi-Line* sub-types if required. Some of the geometry sub-type changing may confuse unwary users, for example, a SpatiaLite Line data source read by QGIS and saved as a SpatiaLite *Multi-Line* is silent and without error or warning, however, the newly created SpatiaLite *Multi-Line* can't be then saved as a SpatiaLite *Line* by QGIS without the use of a tool that changes the geometry type to *Line*, even though each *Multi-Line* only has one part.

Editing existing geometries or creating new geometries creates similar challenges for parsing geometry types. Any geometry edit or creation has to occur on a QGIS geometry sub-type and then parsed to the data source. In an edit process, QGIS will allow incompatible geometries to be created and it is only when the edited geometries are attempted to be committed to the data source that QGIS either throws an error or changes to a compatible geometry sub-type.

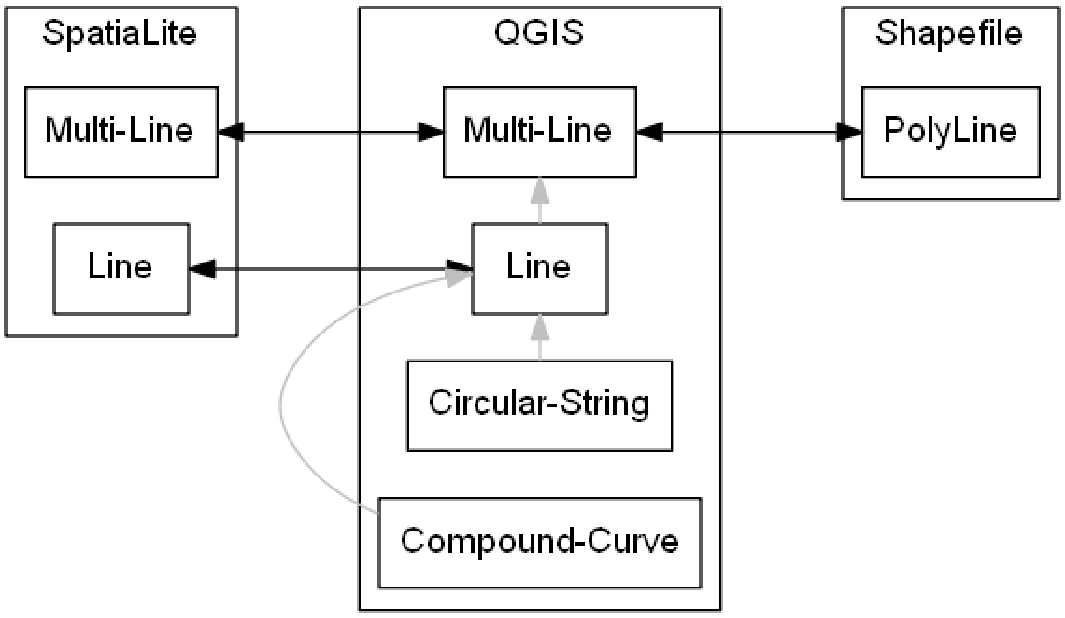


Fig. 4.1: The association of various geometry *Line* sub-types for SpatiaLite, QGIS and Shapefile: grey arrows refer to changes in geometry sub-type within QGIS prior to committing data; black arrows indicate data parsing between QGIS and the external data sources.

[8] The QGIS types were renamed here for clarity. The actual QGIS types are `QgsLineString`, `QgsCircularString`, `QgsCompoundCurve`, and, `QgsMultiLineString`.

4.1.1 Data Provider Geometry Type Errors

There is a great diversity in both the refinement and development of each QGIS data provider. As QGIS is opensource, the varying levels of refinement of different data providers may reflect communities of users or the commissioning of developers. Given the complexity of the data providers task of parsing datasources to and from QGIS, it is not surprising that there are many unexpected errors originating from the data parsing process.

For example, for a *memory* data source, QGIS allows the mixing of primitive and multi-part geometries of the same dimensionality. The insertion of a *Line* geometry subtype record into a *Multi-Line* geometry subtype for a QGIS *memory* data source is demonstrated by Python script using the QGIS API:

```
>>> from qgis.core import *
>>> layerMulti=QgsVectorLayer('MultiLineString?crs=epsg:4326&
↪field=ID:string', 'a', "memory")
>>> providerMulti=layerMulti.dataProvider()
>>> recordWrite = QgsFeature()
>>> recordWrite.setAttributes(['1'])
>>> recordWrite.setGeometry(QgsGeometry.fromWkt('LINESTRING (1 1, 6 1)'))
>>> providerMulti.addFeature(recordWrite)
True
>>> recordRead = layerMulti.getFeature(1)
>>> print(QgsWkbTypes.displayString(recordRead.geometry().wkbType()))
LineString
>>> print(QgsWkbTypes.displayString(layerMulti.dataProvider().wkbType()))
MultiLineString
```

More worryingly, as shown in the next Python script, the reverse is also possible. One may add a feature with a *Multi-Line* geometry sub-type into a *Line* QGIS memory dataset.

```
>>> from qgis.core import *
>>> layerSingle=QgsVectorLayer('LineString?crs=epsg:4326&field=ID:string',
↪'b', "memory")
>>> providerSingle = layerSingle.dataProvider()
>>> recordWrite = QgsFeature()
>>> recordWrite.setAttributes(['1'])
>>> recordWrite.setGeometry(QgsGeometry.fromWkt('MULTILINESTRING ((1 1, 6␣
↪1), (1 2, 6 2))'))
>>> providerSingle.addFeature(recordWrite)
True
>>> recordRead = layerSingle.getFeature(1)
>>> print(QgsWkbTypes.displayString(recordRead.geometry().wkbType()))
MultiLineString
>>> print(QgsWkbTypes.displayString(layerSingle.dataProvider().wkbType()))
LineString
```

Fortunately with a QGIS memory dataset you can't insert a *Point* into a *Line* dataset, or, otherwise mix geometry sub-types of different dimensionality.

```
>>> from qgis.core import *
>>> layerSingle=QgsVectorLayer('LineString?crs=epsg:4326&field=ID:string',
↪'b', "memory")
>>> providerSingle = layerSingle.dataProvider()
>>> recordWrite = QgsFeature()
>>> recordWrite.setAttributes(['1'])
>>> recordWrite.setGeometry(QgsGeometry.fromWkt('POINT (1 1)'))
```

(continues on next page)

(continued from previous page)

```
>>> providerSingle.addFeature(recordWrite)
False
```

4.2 Parsing Empty and null Geometry Data Values

The instantiation process for *empty* and *null* data values in QGIS is very different dependent on whether the value is *empty* or *null*.

4.2.1 Instantiation of Empty Geometry Values by QGIS

An *empty* geometry is a geometry value with an *empty* set of vertices. Hence, an *empty Line* geometry value is distinct to an *empty Multi-Line* geometry value. As the geometry sub-type is associated with an *empty* value, the method for instantiating *empty* geometries is embedded within each geometry sub-class that can be instantiated. Unfortunately, there is some variation with how QGIS instantiates *empty* geometry values across different geometry subclasses.

Using the QGIS API, *empty* geometries for several geometry types can be instantiated by instantiating the relevant QgsAbstractGeometry subclass without a set of vertices. For example, to test that a `QgsLineString()` is *empty*:

```
>>> from qgis.core import QgsLineString
>>> QgsLineString().isEmpty()
True
```

Although *empty* geometries can be created for most geometry types with the QGIS API by instantiation without a set of vertices, it is not currently possible to instantiate an *empty* point geometry using this approach (Table 4.2). As demonstrated below, the well known text representation of the call to instantiate an *empty* point reveals that QGIS is wrongly adding a vertex with the coordinates of (0 0) [8][10].

```
>>> from qgis.core import QgsLineString,QgsPoint
>>> print(QgsLineString().asWkt())
LineString ()
>>> print(QgsPoint().asWkt())
Point (0 0)
>>> print(QgsPoint().createEmptyWithSameType().asWkt())
Point (nan nan)
```

Table 4.2: Testing whether an *empty* geometry has been created by the instantiation of various types of `QgsAbstractGeometry` subclasses using the Python Console in QGIS 3.0.3.

Input	Output
`QgsPoint().isEmpty()`	`False`
`QgsLineString().isEmpty()`	`True`
`QgsPolygon().isEmpty()`	`True`
`QgsGeometryCollection().isEmpty()`	`True`

4.2.2 Changing of Empty data Values by parsing

As indicated in Table 4.1 *empty* geometry values are changed to *null* when parsing to Microsft SQL server or Shapefile data sources. For a Shapefile, this change in data value maybe the most pragmatic

[10] a `Point (nan nan)` is also reported when an *empty* PostGis point is parsed by QGIS.

outcome as *empty* geometries are not included in it's specification. For the Micrsoft SQL Server data source, the change from *empty* to *null* occurs when parsing to or from this data source [28] although both QGIS and Microsoft SQL Server specifications include *empty* geometry data values. In contrast to Microsoft SQL Server, *empty* and *null* values are parsed to and from PostGIS without fault or change.

4.2.3 Inadvertently Creating Empty Geometries within QGIS

Within QGIS, an *empty* geometry is created by using the "Vertex Tool" to delete all vertices of an existing shape. A user unfamiliar with QGIS may incorrectly assume that deleting all the vertices of a geometry also deletes the record. This mistake of assumed record deletion when deleting vertices is most likely to occur for geometries with single part points as the geometry disappears from view when the first and only vertex is deleted. Hence, when editing a shape, a QGIS user may unintentionally create an *empty* shape when they are attempting to delete the shape.

In QGIS null Implies Empty

In the definition of *empty* geometry values in section *Empty* it is articulated that an *empty* geometry is just one valid value in the set of valid values for the geometry data type. In contrast, *null* indicates that the geometry value is unknown and that the value can be *any* value from the set of valid values. Hence, an *empty* geometry values is one of many possible values for a *null* geometry. For example, for a point, the vertices of an *empty* point are $\{\ \}$ as it has no coordinates, and, the vertices of a *null* point are $\{x\ y\}$ where x and y are both variables designating unknown coordinates. These definitions are consistent with SQL relational databases. Unfortunately, the application of *null* used by SQL relational databases is different to that used by both mathematics for set theory, and, by most computer programming languages [6].

Let's consider the definition of *null* for set theory and computer programming using a common example. If one has a box of apples and a box of bananas, then the intersection of the two boxes of fruit is an *empty* box. For set theory, *null* – nothing is what you have when you take away the box.

Similar to set theory, for most object-oriented computer programming languages, an object that has not been instantiated is *null*. Hence, a pointer to something that doesn't exist is *null*. In comparison, an *empty* object is an object that has been instantiated but not populated, for example, a list with no members, or, a box with no fruit.

So, from the perspective of an object-oriented computer programming language, *null* means no value and no type. Hence, a *null* object is an object that does not exist and does not occupy space in a computer's memory. Indeed, SQL relational databases implement *null* as an object that doesn't exist, so, the context difference between unknown and nothing does not have to create conflict.

The underlying conflict is based on logical deduction. It has been proved that an *empty* set is a subset of any set. Some argue that on object that doesn't exist can't contain anything, so, it must be *empty*[11]. Hence, *null* is a subclass of *empty*. This argument is flawed as by arguing that a *null* object is *empty* you have just created an object. A *null* object neither contains anything nor has a container.

Unfortunately, many computer programming languages and QGIS follow a convention where something must also be *empty* if it is *null*. QGIS employs the logic that a *null* geometry is also an *empty* geometry [13]. The QGIS test for *null* in it's C++ code simply asks whether the object exists (whether it points to anything) [13]:

```
bool QgsGeometry::isNull() const
{
```

(continues on next page)

[11] Unable to find a high quality reference for this argument. Several private conversations with C++ programmers reveal that this logic is common and is frequently used to test for *empty* in C++.

(continued from previous page)

```
  return !d->geometry;
}
```

Now, the QGIS test for *empty* in it's C++ code performs the same test as for *null*, and, returns `True` if the test for *null* returns `True` [13]:

```
bool QgsGeometry::isEmpty() const
{
  if ( !d->geometry )
  {
  return true;
  }

  return d->geometry->isEmpty();
}
```

Hence, a test for an *empty* geometry in QGIS will return `True` for all *null* and *empty* geometry values.

```
>>> from qgis.core import QgsFeature
>>> recordNull, recordEmpty = (QgsFeature() for i in range(2))
>>> recordEmpty.setGeometry(QgsGeometry.fromWkt('LINESTRING ()'))
>>> # Only recordNull has a null geometry
>>> recordNull.geometry().isNull()
True
>>> recordEmpty.geometry().isNull()
False
>>> # Both recordNull and recordEmpty have empty geometries
>>> recordNull.geometry().isEmpty()
True
>>> recordEmpty.geometry().isEmpty()
True
```

4.2.4 Instantiation of null Geometry Values by QGIS

To claim a *null* geometry value has been instantiated in QGIS is a *faux pas*. *null* represents the absence of a value, so, a record with a *null* geometry value is a record without a geometry value. *null* infers that an object (in this case a geometry value) has not been instantiated.

4.2.5 Retrospective Incorporation of Empty and null Values Into GDAL

The Geospatial Data Abstraction Library (GDAL) is ubiquitous within most GIS Software to translate and process geospatial data. Handles for *empty* geometries were not part of the original GDAL specification, with *empty* geometries being treated as *null*. The retrospective incorporation of *empty* handles into GDAL has not been picked up by many applications that employ GDAL, and, even within GDAL there are many processing tools that don't maintain *empty* geometries.

Many of the current GDAL set operators continue to convert *empty* geometry values to *null* geometry values even when the set being operated on has successfully implemented the *empty* geometry handle on parsing to GDAL. For example, for the intersection operation the output is *"a new geometry representing the intersection or NULL if there is no intersection or an error occurs"* [4]. Curiously, the same GDAL geometry class has a handle to assign an *empty* geometry, or, to test for an *empty* geometry [1]. On another GDAL ticket register it is stated *"GML/WFS: by default does not advertise NOT NULL fields since we treat empty strings as being null for historical reasons. By setting the open option EMPTY_AS_NULL=NO, empty strings will be reported as such and NOT NULL fields might be* *advertised"* [24]. It is clear that GDAL was not originally intended to distinguish between *null* and *empty*.

The retrospective addition of handles for *empty* geometry values in GDAL was mimicked by QGIS. Consequently, within QGIS there are many tools that convert *empty* values to *null* during their operation.

CHAPTER FIVE

AIM

QGIS is a capable and popular GIS desktop software that changes both *empty* and *null* geometry values, and, geometry type during both data parsing and data processing. Some of these data changes are necessary to comply with different data source requirements. These changes in data are likely to catch recent QGIS adopters unaware and may lead to errors and losses in productivity.

The primary aim of this project is to expose those records with *null* and *empty* geometry values. A secondary aim of this project is to expose the geometry type per record.

If successful, this project will be a stepping stone towards:

- increasing GIS professional awareness of *null* and *empty* geometry values;
- illumination of QGIS data parsing errors when they occur so that developers may address them; and,
- integrating GIS within organizational databases by embracing QGIS's open and flexible data providers.

CHAPTER

SIX

PLUGIN DESIGN

A QGIS plugin was created that generates an attribute table with a geometry describing column for the active vector layer in a QGIS workspace. The created geometry describing column is a mixture of two data types combining both geometry data values and geometry data types. Icons are used for geometry data types to create an intuitive language independent graphical interface.

The plugin is constructed with Python script using Python 3.x. By intention, current QGIS classes and methods are used as much as possible to reduce script complexity and to be resilient to new versions of QGIS. Initially the plugin was created with the QGIS plugin Builder Tool [26].

The main conceptual processes of the plugin are shown in Fig. 6.1 and are described in the following sections.

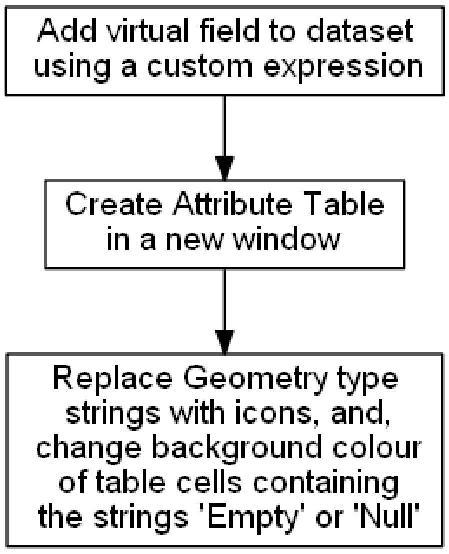

Add virtual field to dataset using a custom expression
Create Attribute Table in a new window
Replace Geometry type strings with icons, and, change background colour of table cells containing the strings 'Empty' or 'Null'

Fig. 6.1: The key processes in the *Geometry Attribute Table* plugin.

6.1 Geometry Describing Expression

The geometry describing expression can be accessed by the *parseQGISGeometry module* in the API, and, is shown below:

```
@qgsfunction(args='auto', group='Custom', usesGeometry=False)
def geometryField(feature, parent):
    #   THIS DOCUMENTATION IS NOT BE SHOWN BY THE SPHINX AUTODOC DIRECTIVE
    """
```

(continues on next page)

(continued from previous page)

```
    Creates a QGIS expression called geometryField.  This expression
    returns a string that represents the geometry in the following
    order of decreasing precedence:

        * Null
        * Empty
        * Well known binary type string

    Returns:
        A string that represents the geometry.
    """
    geom = feature.geometry()
    #   Creates Point(0 0)
    emptyPoint1 = QgsPoint()
    #   Creates Point(nan nan)
    emptyPoint2 = emptyPoint1.createEmptyWithSameType()
    #   Null has to be tested for before empty as QGIS treats all null
    # geometries as empty
    if geom.isNull():
        return 'Null'
    elif geom.isEmpty():
        return 'Empty'
    #   Specific tests for empty points.
    elif geom.type().__eq__(0) and geom.vertexAt(0).__eq__(emptyPoint1):
        return 'Empty'
    elif geom.type().__eq__(0) and geom.vertexAt(0).__eq__(emptyPoint2):
        return 'Empty'
    else:
        return QgsWkbTypes.displayString(geom.wkbType())
```

This expression is performed per individual record (feature). This expression has a string data type output. A few characteristics of this expression are:

1. it needs to test for *null* geometry values before *empty* geometry values;
2. it has specific tests for *empty* point geometry values as these are given the set of vertices *{0 0}* or *{nan nan}* by QGIS depending on how they are instantiated;
3. It reports the feature's geometry data value when the geometry is *empty* or *null*, otherwise, it reports the feature's geometry type; and,
4. in the output, geometries with elevation (z values) or with measurements (m values) will still be distinguished from those without either.

This expression is used to populate a virtual field called *"Geometry"* that is appended to a vector dataset. A virtual field is a computed field that exists in memory only and is not written to the datasource. As the virtual field is appended to the vector layer, it may be viewed or used wherever a virtual field is present in QGIS. For example, when the window created by the plugin is open, the created *"Geometry"* field is included as a column in the standard *Attribute Table*.

6.2 Attribute Table Creation

The attribute table shown by the plugin is created by a process that is analogous to the standard QGIS Attribute Table. Basically, a cache of the dataset is made, and a model of the dataset is constructed so that Qt's model-view [11] approach can be applied. When a piece of software outputs a table display of a data source, the software needs to replicate each data item and keep a synchronization between the original and the replication. Qt's model-view reduces both the replicated dataset's size and reduces the computing resources required for synchronization between the primary and replicated datasets. The following script is contained within the `run(self)` method that is within the `geomAttribute` class in the `geomAttributeRun` module of this plugin:

```
#   Create a cache of the vector layer data of size 10000
#   Cache geometry is true by default
#   For example: print(self.vector_layer_cache.cacheGeometry())
self.vector_layer_cache = QgsVectorLayerCache(self.layer, 10000)

#   QgsAttributeTableModel is a subclass of QAbstractTableModel
self.attribute_table_model = QgsAttributeTableModel(self.vector_layer_
↪cache)
```

Within the same `run(self)` method of the `geomAttribute` class a filter is applied to the model of the dataset to cater for situations where selections are made:

```
#   The QgsAttributeTableFilterModel() is used to synchronize any␣
↪selection.
self.attribute_table_filter_model = QgsAttributeTableFilterModel(
self.iface.mapCanvas(), self.attribute_table_model)
```

6.3 Putting Icons into the Attribute Table

Qt's item delegate function is used to replace the string field in the virtual *"Geometry"* column in the plugin's Attribute Table. This delegate function does not affect the same virtual field where it is used outside of the plugin's Attribute Table. For example, the output of the same virtual field in the standard QGIS Attribute Table remains a string. Icon's only replace strings for geometry types that *contain* the strings listed in Table 6.1 which are contained within a Python dictionary in the script. Any geometry type that doesn't contain a string shown in Table 6.1 will remain as a string. For example, a string value of *CircularString*, *CompoundCurve* or *GeometryCollection* will remain unchanged and not be replaced by an icon. As the script searches for strings that *contain* the string: the string-switched-icon represents all variations of a geometry subtype including those that contain elevations or measurements; and, the order of searching and hence the order of dictionary keys is essential. For example, a search for *Point* would return *True* for *MultiPoint* as the string *Point* is contained within *MultiPoint*. So, *MultiPoint* needs to be searched for prior to *Point*.

Strings instead of icons are only used for *empty* and *null* geometry values with the application of background colors for several reasons.

For *empty*, there are several notations to represent an *empty* set including $\{\ \}$ and $\varnothing$. Neither of these notations for *empty* sets are widely known outside of mathematics and many QGIS users may not understand them without investigating. To aid interpretation, the string *empty* is kept in the Attribute Table with a background color of dark yellow. The use of tooltips for *empty* geometry data values is one possible approach in the future to use the empty set notation of $\varnothing$ instead of the string *empty*.

One possible symbol for *null* is x which normally refers to an unknown variable, but, *null* refers to an unknown set of variables, not a single variable. Imieliński and Lipski proposed the use of *"@"* for *null* [19], however, this was not adopted by other researchers. As there is no universal symbol for *null*, the string *null* was used in combination with a background color of red.

Table 6.1: The strings that are replaced with icons by the item delegate function.

Precedence	Geometry Type
1	MultiPoint
2	Point
3	MultiLine
4	Line
5	MultiPolygon
6	Polygon

The Python script that exchanges geometry type strings with icons, and, changes the background color of *Empty* and *null* geometry value strings is contained within the `paint` method in the `myDelegate` class of the `geomAttributeRun` module, as shown below:

```
    def paint(self, painter, option, index):
        """
        This method paints the geometry column attributes with icon's for
        display in the Plugin's Attribute-Table window.

        Args:
           option(QStyleOptionViewItem):
           painter(QPainter):
        """

        value = index.data(Qt.DisplayRole)

        iconDict = {
            "MultiPoint": "point_2x",
            "Point": 'point_1x',
            "MultiLine": 'line_2x',
            "Line": 'line_1x',
            "MultiPolygon": 'polygon_2x',
            "Polygon": 'polygon_1x'
        }

        #   Equivalent to "for iconKey in iconDict:"
        geometry = next(filter(value.__contains__, iconDict.keys()), None)
        if geometry is not None:
            icon = QIcon(':/plugins/geomAttribute/icons/{}.png'.
↪format(iconDict.get(geometry)))
            icon.paint(painter, option.rect, Qt.AlignCenter)
        else:
            #   Set background color
            painter.setPen(QPen(Qt.NoPen))
            backgroundColor = Qt.lightGray
            geometry = next(filter(value.__eq__, ['Null', 'Empty']), None)
            if geometry is not None:
                if value.__eq__('Null'):
                    backgroundColor = Qt.red
                else:
                    backgroundColor = Qt.darkYellow
            painter.setBrush(backgroundColor)
            painter.drawRect(option.rect)
            #   Set text color - order is important.
```

(continues on next page)

(continued from previous page)

```
            #   If done before background color will not show.
            painter.setPen(QPen(Qt.black))
            painter.drawText(option.rect, Qt.AlignCenter, value)
```

6.4 Plugin Window

The attribute table produced by the plugin is contained within a Qt Window. The preliminary plugin created by the QGIS *Plugin Builder* plugin instantiated a Qt Dialog. This Qt Dialog was exchanged with a Qt Window during the development of this plugin so that a tool bar with a help icon could be added to the window containing the attribute table. The Qt Window allows for further development of the plugin.

The plugin window is instantiated by the `geomAttribute_window` module:

```
class geomAttributeWindow(QtWidgets.QMainWindow, FORM_CLASS):
    def __init__(self, parent=None):
        """
        Constructor
        """
        super(geomAttributeWindow, self).__init__(parent)
        #   Set up the user interface that is constructed with Qt Designer.
        self.setupUi(self)
```

The plugin window design is in Qt's designer file format (file extension .ui). With this approach, the window design can be changed independently from the plugin's Python script using Qt Designer. To function, the *"Table View"* widget in Qt Designer had to be subclassed with the QgsAttributeTableView.

As Qt's item delegate is used to replace strings in the produced attribute table with icons, all of the filenames for the icons had to be added to the Qt resource collection file (`resources.qrc`). The resources used by Qt have to be compiled whenever changes are made to the resource collection file.

The instantiation of the *Geometry* virtual field is coupled to the plugin's window instantiation, such that this field is deleted when the plugin's window is closed.

6.5 Plugin Help Files

The initial intention was to contain all of the plugin's help files within the plugin so that they can be accessed offline. However, the construction of help files using the Qt framework was cumbersome. It was more efficient to produce HTML help files using Sphinx[15] and have these hosted by the plugin's GitHub site. An inspection of the help files for QGIS[16] and the help files of many QGIS plugins revealed that most QGIS plugin help files are hosted on the world wide web and require an internet connection to access. Ironically, Sphinx can also produce *Qt Help* formatted files, and, this may be exploited in the future to generate offline help files.

6.6 Future Development

There is much scope to enhance the plugin. Some priorities for improvement are:

1. add a summary table that reviews large datasets for *null* or *empty* geometry values;

[15] http://www.sphinx-doc.org/en/master/
[16] https://docs.qgis.org/testing/en/docs/user_manual/index.html

2. add another virtual field that displays the number of parts for multipart geometries;
3. add icons for geometries with curved segments; and,
4. add support for geometries with measurements or elevation.

CHAPTER

SEVEN

TOOLS INCLUDED WITH THIS PLUGIN

A QGIS 3.x plugin that shows an *Attribute Table* with a geometry describing column for the active vector layer.

This plugin has the following tools:

7.1 Geometry Attribute Table

An attribute table window which includes a geometry describing column for any vector layer.

The geometry column uses the icons shown in Table 7.1 to indicate a feature's geometry value or type in descending precedence: unknown (null); empty; point; multi-part point; line; multi-part line; polygon; and, multi-part polygon.

Table 7.1: Icons used to represent various geometries.

Icon	Geometry Description	Precedence
Null	Unknown value	First
Empty	Empty value	Second
+	Point type	Third
+ +	Multi-Part Point type	Fourth
/	Line type	Fifth
/ \	Multi-Part Line type	Sixth
○	Polygon type	Seventh
□ ○	Multi-Part Polygon type	Eighth

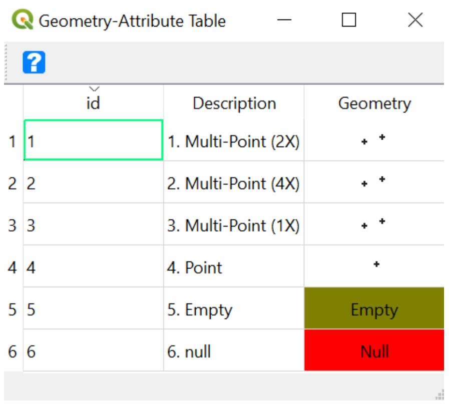
The plugin does not distinguish between two dimensional and three dimensional shapes, nor between those with or without measurements. The plugin does not consider whether geographic or cartesian coordinates are used for the spatial reference system.

7.1.1 Use

This tool works on the current active layer within QGIS. The current layer needs to be a vector layer. This tool will not work on a raster layer.

Within QGIS, click on the vector layer to be analysed in the *Layers Panel* to make it active, then click on the *Geometry Attribute Table* icon or select the *Geometry Attribute Table* item from the *Geometry*

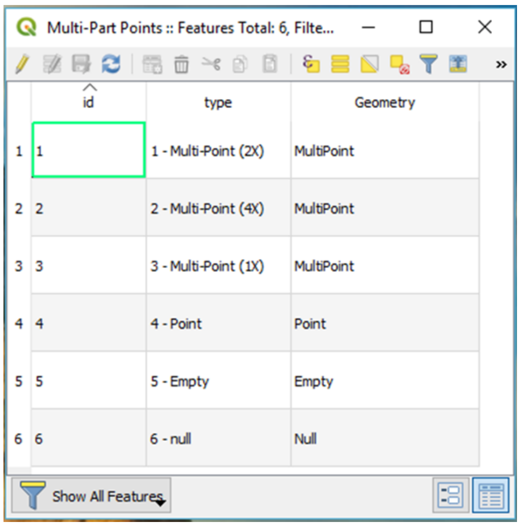
Attribute Table group on the *Plugins* menu. Successful use of the *Geometry Attribute Table* tool will display a new window with the title *"Geometry-Attribute Table"* that contains an attribute table where the last column has the heading *"Geometry"*. Fig. 7.1 shows an example of an open *Geometry Attribute Table* output window with a *"Geometry"* column heading. This geometry describing column uses the icons shown in Table 7.1 to represent each record's geometry.


Geometry-Attribute Table

	id	Description	Geometry
1	1	1. Multi-Point (2X)	⁺ ⁺
2	2	2. Multi-Point (4X)	⁺ ⁺
3	3	3. Multi-Point (1X)	⁺ ⁺
4	4	4. Point	⁺
5	5	5. Empty	Empty
6	6	6. null	Null

Fig. 7.1: A screen shot of the window containing an attribute table produced by the *Geometry Attribute Table* tool in a Microsoft Windows environment.

A text version of the *Geometry* column will also be appended to the conventional QGIS attribute table (Fig. 7.2). Successful closing of the *Geometry Attribute Table* window will also remove this appended *Geometry* column.


Multi-Part Points :: Features Total: 6, Filte...

	id	type	Geometry
1	1	1 - Multi-Point (2X)	MultiPoint
2	2	2 - Multi-Point (4X)	MultiPoint
3	3	3 - Multi-Point (1X)	MultiPoint
4	4	4 - Point	Point
5	5	5 - Empty	Empty
6	6	6 - null	Null

Fig. 7.2: A screen shot of the standard QGIS Attribute Table window showing the same dataset shown in Fig. 7.1 in a Microsoft Windows environment. The column with the heading *"Geometry"* is a virtual field created by the *Geometry Attribute Table* tool and will be removed when the *Geometry-Attribute Table* window is closed.

7.2 Load Data

The *Load Data* function creates the following QGS memory vector layers and adds them to the current QGIS work space:

- Multi-part Lines

- Multi-part Points

Both of these layers contain records with: primitive geometries; multi-part geometries; *empty* geometry values; and, *null* geometry values. These layers are ideal to observe the utility of the *Geometry Attribute Table* tool.

The methods that create these layers are contained within the plugin's *modelVectorLayers module*.

7.2.1 Use

Select the *Load Model Data* item from the *Geometry Attribute Table* group on the QGIS *Plugins* menu.

CHAPTER

EIGHT

API

The following modules are included within this Plugin. The Plugin's repository is on GitHub[17].

8.1 modelVectorLayers module

Creates vector layers with a QGIS memory layer data source. These layers are intended to demonstrate how the Geometry Attribute Table plugin renders datasets that have: multi-part, single part, *empty*, and, *null* geometries. A description of the created layer is included with each method.

`modelVectorLayers.`**`createMultiLines`**`(`*layerName='Multi-Part Lines'*`)`

A geometry collection of multi-part lines and lines (single part constrained).

Args: layerName (str): The name of the layer that is loaded into QGIS.

Returns: A memory vector layer containing features with the following geometries: 2 part multi-line; 3 part multi-line, 1 part multi-line, line, Empty multi-line, and, Null.

`modelVectorLayers.`**`createMultiPoints`**`(`*layerName='Multi-Part Points'*`)`

A geometry collection of multi-part points and points (single part constrained).

For an *empty* point, QGIS creates erroneously creates a point with the coordinates (0 0).

Args: layerName (str): The name of the layer that is loaded into QGIS.

Returns: A QGIS memory data source vector layer containing features with the following geometries: 2 point multi-point; 4 point multi-point, 1 point multi-point, point, *empty* multi-point, and, *null*.

8.2 parseQGISGeometry module

The *parseQGISGeometry* module includes a QGIS expression called **geometryField** that returns a string that represents a vector feature's geometry. The **geometryField QGIS expression** is described in detail in the section *Geometry Describing Expression*.

The QGIS Expression and methods in this module parse each record's vector geometry for a QGIS vector layer.

`parseQGISGeometry.`**`layerAddVirtualGeometryField`**`(`*vectorLayer*`)`

Appends a virtual field called *"Geometry"* to the input *vectorLayer*. This virtual field consists of string values populated by the *"geometryField"* expression that is contained within this module.

Args: vectorLayer (QgsVectorLayer): A QGIS vector layer.

[17] https://github.com/PhilipWhitten/geomAttribute

`parseQGISGeometry.`**`layerRemoveVirtualGeometryField`**`(`*vectorLayer*`)`

Removes the virtual field called *'Geometry'* from a vector layer if it exists.

Args: vectorLayer (QgsVectorLayer): A QGIS vector layer.

`parseQGISGeometry.`**`layer_review`**`(`*layer*`)`

IN DEVELOPMENT - NOT CURRENTLY USED BY PLUGIN

Reviews any vector layer for the presence of *Null* or *Empty* geometries. Provides a summary of the vector geometry types that occur within a layer.

Args: vectorLayer (QgsVectorLayer): A QGIS vector layer.

8.3 zipGeomAttribute module

Contains methods for zipping the minimum compiled source scripts needed for this plugin to function within QGIS. These methods do not do any of the compilation (e.g. update documentation, or, compile the QT Designer interface).

`zipGeomAttribute.`**`installGeomAttributeFromZip`**`(`*pluginPath*`)`

Unzips the zipped plugin folder. An automation used for testing.

Args: pluginPath (str): The path of the plugin folder in the user's installation of QGIS.

`zipGeomAttribute.`**`zipGeomAttribute`**`()`

Zips the minimum number of files needed for the plugin to be installed into QGIS

CHAPTER

NINE

CHANGELOG

This project adheres to Semantic Versioning[18].

9.1 Version 0.1.0

- First release
- Shows attribute table for vector layers in QGIS v3.x with icons that represent the geometry of each record.
- Has the option to create demonstration vector datasets.

9.2 Version 0.1.1

- Removes the virtual field called *"Geometry"* when the created *"Geometry-Attribute Table"* is closed.

[18] https://semver.org/spec/v2.0.0.html

BIBLIOGRAPHY

[1] Empty(self, *args). URL: http://gdal.org/python/osgeo.ogr.Geometry-class.html#Empty.

[2] How To : Insert a NULL or empty st _ geometry attribute in Oracle. URL: https://support.esri.com/en/technical-article/000010466.

[3] QGIS - The Leading Open Source Desktop GIS. URL: https://www.qgis.org/en/site/about/index.html#.

[4] def Intersection(self, *args):. URL: http://gdal.org/python/osgeo.ogr-pysrc.html#Geometry.Intersection.

[5] ISO 19125-2 Geographic information - Simple feature access. Part 2: SQL Option. First Edition. 2004.

[6] What Is Null. 2010. URL: http://wiki.c2.com/?WhatIsNull.

[7] Spatial Data Types Overview | Microsoft Docs. 2016. URL: https://docs.microsoft.com/en-us/sql/relational-databases/spatial/spatial-data-types-overview?view=sql-server-2017.

[8] Bug report #19190: QgsPoint class does not store empty point geometries - QGIS Application - QGIS Issue Tracking. 2018. URL: https://issues.qgis.org/issues/19190.

[9] Changelog for QGIS 3.0. 2018. URL: https://www.qgis.org/en/site/forusers/visualchangelog30/.

[10] INTERSECT | meaning in the Cambridge English Dictionary. 2018. URL: https://dictionary.cambridge.org/dictionary/english/intersect.

[11] Model/View Programming | Qt Widgets 5.12. 2018. URL: http://doc.qt.io/qt-5/model-view-programming.html.

[12] QGIS API Documentation: QgsAbstractGeometry Class Reference. 2018. URL: https://qgis.org/api/classQgsAbstractGeometry.html.

[13] QGIS API Documentation: Source File. 2018. URL: https://qgis.org/api/qgsgeometry_8cpp_source.html.

[14] QGIS/src/providers/. 2018. URL: https://github.com/qgis/QGIS/tree/master/src/providers.

[15] Tutorial: Perform web editing using data from a database—Documentation | ArcGIS Enterprise. 2018. URL: http://enterprise.arcgis.com/en/server/latest/get-started/windows/tutorial-performing-web-editing-using-data-from-a-database.htm.

[16] Wikipeday Contributors. QGIS. URL: https://en.wikipedia.org/wiki/QGIS.

[17] Wikipedia Contributors. Set (mathematics). 2018. URL: https://en.wikipedia.org/w/index.php?title=Set_(mathematics)&oldid=866126059.

[18] ESRI. ESRI Shapefile Technical Description. Technical Report, ESRI, 1998. URL: www.esri.com,.

[19] Tomasz Imieliński and Witold Lipski Jr. Incomplete Information in Relational Databases. *J. ACM*, 31(4):761–791, sep 1984. URL: http://doi.acm.org/10.1145/1634.1886, doi:10.1145/1634.1886[19].

[20] Mateusz Loskot. SqlGeometry and POINT EMPTY in WKB. 2010. URL: http://mateusz.loskot.net/post/2010/02/26/sqlgeometry-and-point-empty-in-wkb/.

[21] OGC. OpenGIS\textregistered Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture. *Open Geospatial Consortium, Inc*, pages 93, 2010. URL: http://portal.opengeospatial.org/files/?artifact_id=25355E+Implementation+Standard+for+Geographic+information+-+Simple+feature+access#1.

[22] Paul Ramsey. Nothing, Nada, Zip, Bupkus. 2010. URL: http://blog.cleverelephant.ca/2010/03/nothing-nada-zip-bupkus.html.

[23] Anthony Robinson. Representing the Presence of Absence in Cartography. *Annals of the American Association of Geographers*, pages 1–15, feb 2018. URL: https://www.tandfonline.com/doi/full/10.1080/24694452.2018.1473754, doi:10.1080/24694452.2018.1473754[20].

[24] Even Rouault. Empty elements in GML : empty or NULL. 2015. URL: https://trac.osgeo.org/gdal/ticket/5968.

[25] Ron van der Meyden. Logical Approaches to Incomplete Information: A Survey. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*, chapter Logical Ap, pages 307–356. Kluwer Academic Publishers, Norwell, MA, USA, 1998. URL: http://dl.acm.org/citation.cfm?id=294135.294145.

[26] Gary Sherman. Plugin Builder Tool. URL: http://g-sherman.github.io/plugin_build_tool/.

[27] Gary Sherman. History of QGIS Committers. 2011. URL: http://spatialgalaxy.net/2011/09/23/history-of-qgis-committers/.

[28] Philip Whitten. Bug report #19397: Parssing of empty geometry values from MicroSoft SQL Server - QGIS Application - QGIS Issue Tracking. 2018. URL: https://issues.qgis.org/issues/19397.

[19] https://doi.org/10.1145/1634.1886
[20] https://doi.org/10.1080/24694452.2018.1473754