Towards the Internet of Water: Using graph databases for hydrological analysis on the Flemish river system

Erik Bollen[1,2] | **Rik Hendrix**[2] | **Bart Kuijpers**[1] | **Alejandro Vaisman**[3]

[1]Hasselt University, Databases and Theoretical Computer Science Research Group and Data Science Institute, Agoralaan, Gebouw D, Diepenbeek, 3590, Belgium

[2]Data Science Hub, Vlaamse Instelling Voor Technologisch Onderzoek (VITO), Boeretang 200, Mol, 240, Belgium

[3]Instituto Tecnológico de Buenos Aires, Departamento de Informática, C1002ABJ, Buenos Aires, Argentina

Correspondence
Bart Kuijpers, Hasselt University, Databases and Theoretical Computer Science Research Group and Data Science Institute, Agoralaan, Gebouw D, 3590 Diepenbeek, Belgium.
Email: bart.kuijpers@uhasselt.be

Abstract

The "Internet of Water" project will deploy 2,500 sensors along the Flemish river system, in Belgium. These sensors will be part of a monitoring system. This will produce an enormous amount of data, on which prediction and analysis tasks can be performed. To represent, store, and query river data, relational databases are normally used. However, this choice introduces an "impedance mismatch" between the conceptual representation (typically a graph) and the storage model (relational tables). To solve this problem, this article proposes to use graph databases. The Flemish river system is presented as a use case and the Neo4j graph database and its high-level query language, Cypher, are used for storing and querying the data, respectively. A relational alternative is implemented over the PostgreSQL database. A collection of representative queries of interest for hydrologists is defined over both database implementations.

1 | INTRODUCTION AND MOTIVATION

Recent climate changes are increasingly leading to extreme meteorological weather phenomena. These situations affect water supply and water quality, for example, due to the influence of the salty sea on rivers, which can have a negative impact on the water and surrounding land area. Monitoring water quality and quantity is becoming more and more relevant. In Belgium, the "Internet of Water" project (https://www.internetofwater.be/wat-is-internet-of-water/) (IoW) aims to enhance monitoring and governance of the Flemish waterways. It plans to deploy 2,500 sensors along the Flemish river system. These will allow, for example, a warning to be triggered if certain measurements exceed pre-defined thresholds. To take a second example, if a pollution problem is detected by a

sensor at a certain location, the state at downstream locations could be predicted in order to take timely appropriate action. Also, typical data analysis tasks can be performed using the enormous amount of data that will be produced. All of the above requires appropriate modelling, storing and querying of such data. Normally, this would be done using relational databases. Nowadays, graph databases are also good candidates for these tasks, as the following discussion suggests.

Property graphs (Robinson, Webber, & Eifrém, 2013), that is, graphs whose nodes and edges are annotated with properties, are typically used to model networks (e.g., social networks, sensor networks) to perform data analysis. The property graph data model is an abstraction that can also be used to represent rivers in a natural way. For example, using this model, the river segments can be represented as nodes in a graph, and an edge would go from one segment to another if they are consecutive in the direction of the flow. In addition, spatiotemporal coordinates can be included as properties, as well as other characteristics of the river segments. Also, hierarchical contextual data could be defined which would allow the graph to be represented at different granularities, for analytical querying involving data summarisation. Modelling rivers using graphs allows them to be stored in a natural way, using graph databases (Angles, 2012, 2018), rather than relational databases, preventing the "impedance mismatch" problem that arises when the natural network structure is split into many records of a relational table. For example, when a river network is stored as a graph, and represented as indicated above, finding a path between river segments is straightforward using a native graph database.[1] On the other hand, using a relational database, segments would be represented as rows in a table, therefore, finding a path requires self-joining the table as many times as the length of the path requires. In particular, in this article, the Neo4j graph database (http://www.neo4j.com) is used. Besides its popularity, the Neo4j community has developed several libraries of functions which can easily be added to the database as plugins. These libraries include a powerful machinery of algorithms for finding paths in graphs, handling many different data types and performing standard data science tasks. There is also a spatial library (https://github.com/neo4j-contrib/spatial-algorithms/releases/tag/0.2.3-neo4j-4.1.3), which can enhance the analysis possibilities. Last, but not least, Neo4j comes with a high-level graph query language, Cypher.

This article proposes the use of graph databases for facilitating the work of hydrologists along two main dimensions. On the one hand, certain queries of interest can be expressed intuitively by non-expert professionals. On the other hand, more involved queries may sometimes run faster on the graph database than on the relational alternative, thanks to specialised native data structures that allow efficient path traversal. Concretely, the work tackles the following questions. First, can graph databases be successfully used to model, store, and query river flows? Second, if so, what kinds of queries could benefit the most from this approach? And third, is it simpler and more intuitive for a non-expert user to express queries using high-level graph query languages than by writing SQL queries on a relational database? To answer these questions, the Flemish river system is studied and discussed in depth. Furthermore, the process of transforming the source data into a format suitable for querying is also addressed in this work.

In summary, the contributions of this article are: (a) the definition of a property graph data model for representing river systems, which can be extended to other kinds of transportation networks; (b) a real-world case study of this proposal, using the complete Flemish river system; and (c) a description of the data acquisition and transformation processes, which take the river system data from a shapefile into a relational database, create a graph and store it using graph databases; and a definition and analysis of a collection of queries, expressed in Cypher and SQL, and executed on Neo4j and PostgreSQL databases, respectively. The queries are run and the results discussed and reported. It follows from the experiments and analysis that, in most cases, queries on the graph database show better performance (with a few exceptions) than their relational equivalent, particularly in the queries asking for paths. Also, in many cases, queries are more easily and naturally expressed in Cypher than in SQL. However, for some queries, good performance is achieved at the expense of writing more complex Cypher expressions, which are not very intuitive.

The remainder of this article is organised as follows. In Section 2 related work is discussed. The problem of acquiring and preparing the river data is discussed in Section 3, and the relational and graph storage are described

and discussed in Section 4. A case study is presented in Section 5 where a collection of queries are proposed, to analyse the data in relational and graph databases. An experimental evaluation of these queries, in Neo4j and PostgreSQL, is reported and discussed in Section 6. Finally, Section 7 addresses future work and open problems, and concludes.

2 | RELATED WORK

This section studies related work, starting from a description of the context of the problem, namely the rivers in the Flanders region of Belgium. Then graph databases are discussed. Finally, a brief comparison between relational and graph databases is presented.

2.1 | Data-driven approaches to studying flows in river systems

The region of Flanders is located in the northern part of Belgium. In spite of encompassing a relatively small area, watersheds within Flanders exhibit a wide range of regimes which require localised parameterisations, for more accurate hydrological modelling (Heuvelmans, Muys, & Feyen, 2004). In recent decades, the probability of extreme meteorological events has increased in Belgium. This includes the occurrence of heavy storms and frequent heatwaves, resulting in increased incidence of floods and drought periods (Brouwers, Peeters, Van Steertegem, & Van Lipzig, 2015). Drier periods, specifically, have a dual impact in the region, as less rainwater runoff causes higher risks of seawater intrusion from the North Sea, resulting in salinisation of groundwater and soils. As more than 50% of the area of Flanders is used for agriculture, such events severely impact the country's socioeconomic status (Gobin, 2012). The implementation of a dense sensor network over a hydrologically complicated and environmentally vulnerable region allows an integrated geospatial data-driven river system to be built. To put the problem in context, Figure 1 shows an overview of the Flemish river system, using QGIS (https://www.qgis.org) on an OpenStreetMap background. The vast network of river branches can be clearly seen.

Physical process-based modelling, as described above, although necessary, does not suffice for the currently vast amounts of data from various sources for real-time applications. Additionally, commonly used spatially

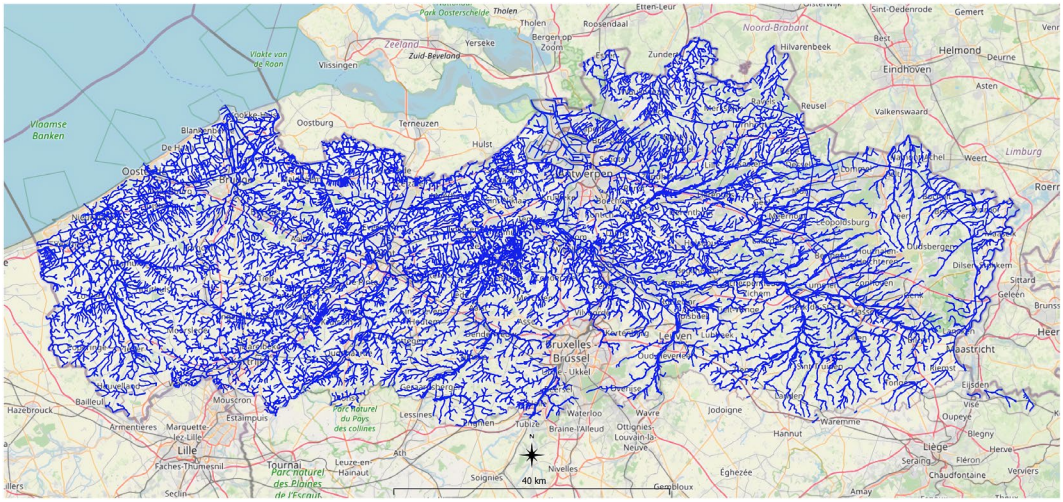

FIGURE 1 Overview of the Flemish river system

distributed hydrologic models still rely, to some extent, on empirical parameterisations and extensive calibration. The implementation of complementary data-driven approaches has become increasingly popular and has successfully represented hydrological processes (Ahani, Shourian, & Rahimi Rad, 2018; Solomatine & Ostfeld, 2008). Data-driven approaches allow for additional insights based on classifications or clustering of regions with similar input and output relationships at varying spatial or temporal resolutions, a task not easily implemented in traditional process-based models.

Typically, relational databases are considered as the standard systems for storing data like that needed for the study introduced above. However, as argued in Section 1, graph databases are natural candidates for the task, since the river topology can be modelled as a graph and stored in native data structures appropriate for answering the required queries efficiently. Demir and Szczepanek (2017) extensively discuss graph data models as a natural way of representing river networks. In fact, they simplify the analysis, arguing that a *tree* representation would suffice to cover a hydrologist's interests. The present article shows (see Section 5, Query 5.7) that this simplification is not very realistic. Also, benchmarking was performed on a small tree containing 1,000 nodes, and using PostgreSQL to store the graph data model. That is, the approach does not tackle the discrepancy between the conceptual and physical data models. Daltio and Medeiros (2015) address this issue, proposing using Neo4j to store a river network. They present Hydrograph, a tool for exploring geographic data in graph databases. The graph data model adopted in this work is a binary tree. However, the work does not report implementation details, or queries on the river system.

We note that both cases discussed above simplify the problem by adopting a tree representation. In contrast, in the present article an actual graph representation is assumed, which, as will be discussed below, is much more demanding, since it increases the number of possible paths, but, on the other hand, realistically represents the river flow. In addition, no tests on graph databases are reported in such efforts. In this article, relational databases (in this case, PostgreSQL) and graph databases (Neo4j) are tested under a collection of typical queries required in river analysis and parameter prediction.

Relational database technology is mature and well known, while graph databases are relatively new, therefore a brief description is provided next.

2.2 | Graph databases

In the context of graph databases, two models are used in practice: (a) models based on RDF (https://www.w3.org/RDF/), oriented to the Semantic Web; and (b) models based on property graphs. Models of type (a) represent data as sets of triples where each triple consists of three elements that are referred to as the subject, the predicate, and the object of the triple. These triples allow arbitrary objects to be described in terms of their attributes and their relationships to other objects. Informally, a collection of RDF triples is an RDF graph. In models of type (b) (Angles et al., 2017), nodes and edges are labelled with a sequence of attribute–value pairs. This is an extension of classical graph database models, frequently used for implementations in practical applications. The main reason for storing attributes in nodes and edges is to speed up the retrieval of the data directly related to a certain node. For an extensive and comprehensive bibliography on graph database models, the interested reader is referred to Angles and Gutierrez (2008) and Angles (2018). Although the models of type (a) have a general scope, the structure of RDF makes them not as efficient as the other models, which are aimed at reaching a local scope. An important feature of RDF-based graph models, however, is that they follow a standard, which is not yet the case for the other graph databases, therefore they are typically used for metadata representation. Many works have proposed RDF to annotate trajectories with semantic information (da Silva, Times, & Renso, 2015; Fileto et al., 2015; Ruback, Casanova, Raffaetà, Renso, & Vidal, 2016). Hartig (2014) proposes a formal way of reconciling both models through a collection of well-defined transformations between property graphs and RDF graphs. He shows that property graphs could, in the end, be queried using SPARQL (https://www.w3.org/TR/

rdf-sparql-query/), the standard query language for the Semantic Web. The model used in the next sections to represent and query trajectory data is based on the concept of property graphs.

Several data models to perform analytical queries on graphs have been proposed. GraphOLAP (Chen, Yan, Zhu, Han, & Yu, 2009), conceptually, is a framework for online analytical processing (OLAP) on a set of homogeneous graphs, based on splitting the graph into a collection of snapshots that are aggregated in two ways, called informational and topological OLAP aggregations. Graph Cube (Zhao, Li, Xin, & Han, 2011) provides a framework for computation and analysis on OLAP cubes using the different levels of aggregation of a graph. Gómez, Kuijpers, and Vaisman (2019) use graph databases to represent semantic trajectory data based on places of interest (PoIs), that is, a collection of trajectories represented as routes between context-defined POIs rather than actual geographic points (Parent et al., 2013).

2.3 | Graph and relational databases

The comparison between relational databases and graph databases has been studied to a limited extent, given that graph database technology is relatively novel. Vicknair et al. (2010) compare MySQL and Neo4j through a simple database schema and relatively simple queries. A similar study was carried out by Batra and Tyagi (2012), also using MySQL and early versions of Neo4j. Both studies, however, discuss very simple queries. Regarding spatiotemporal data, Makris (2019) compare MongoDB, a document NoSQL database, against PostgreSQL, not only for querying but also for data preparation tasks. Gómez et al. (2019) compare graph and relational databases for storing and querying trajectory data, concluding that in most queries, the former perform better because they take advantage of the native data storage, in particular for path traversal. The latter is the only study that compares both models for queries that can exploit the natural representation of the model at hand. The present article works along the same lines, since the river system representation is naturally a network, which can benefit from the native graph data storage of Neo4j in particular, and graph databases in general. The study is presented in Section 4.

3 | DATA ACQUISITION AND PRE-PROCESSING

This section details the data sources used in the article and the pre-processing work carried out in order to prepare the data. The process includes several non-trivial steps that are worth discussing. First, the data sources are described. Then the process that transforms the data into a graph containing the river system information is studied in detail.

3.1 | Data sources

The Flemish environmental agency (VMM) produces the “Vlaamse Hydrografische Atlas” (VHA), a data set comprised of shapefiles containing all the rivers in Flanders, and the watersheds the rivers are part of. This data set does not contain ponds and other water bodies. The VHA is maintained by the VMM, and new versions are released every 3 months. The data set contains geometric data where the rivers in Flanders are represented as line segments, and includes the flow direction of each segment. The main attributes in the data set are (the names of the properties are in Dutch) as follows:

- `vhas`, a unique number that each river segment is assigned by the VMM. This number can be seen and used as an ID for the segment.

- `catc`, the category to which the segment belongs. All rivers are divided into categories, which range from 0 to 9, with 0 representing the biggest waterways and 9 the smallest ones.
- `lengte`, the (pre-computed) length of the segment.
- `geom`, the geometry of the river segment. Most of the time, this is a multi-line (polygonal) geometry.
- `naam`, the name of the river.
- `strmgeb`, the name of the catchment area.
- `beknaam`, the name of an administrative subarea of the catchment. This can be seen as a broad drainage area.
- `lblkwal`, the intended quality of the water in the segment, for example “drinkable water”.

The VHA data set includes more properties of the segments, not included here for the sake of space, but included in the databases that are created for this work. All properties and their description can be found the documentation supplied along with the shapefiles. Additionally, OpenStreetMap information is used, since it is considered here as correct and up-to-date, in general, for Belgium (https://openstreetmap.be/en/). Specifically, for the tests reported here, the VHA data set for 7 August 2020 is used (http://www.geopunt.be/catalogus/datasetfolder/020a452d-8cd2-41b7-9c64-2be367668837).

3.2 | Preparing the data set

The VHA described in the previous section must be processed to produce data that can be used for analysis and prediction. This process is comprised of two steps: first, create the relationships between river segments; and second, fix the errors that may have occurred.

3.2.1 | Creating relationships between river segments

The representation of the overall water flow must be added to the data set, since the data contain the flow direction within each segment, but not over multiple segments. A new relation is defined encoding this overall flow information. The terminology of the segments needs to be defined first. When water is flowing from one segment to the next, the two segments involved are called `source` and `target`, respectively. The former (latter) is the segment where the water is coming from (flowing to). In other words, it can be said that the target segment follows the source segment for downstream flows. Having named the two segments involved, a relation `flows-to(A,B)` can be defined as a binary relation where A and B are the IDs of the segments. The relation consists of all tuples (a, b) where a is a source segment ID and b is a corresponding target segment ID.

In order to create this relation, each segment has to be matched with all the other segments, to check whether or not the water flows directly from one segment to the other. The main idea is that the endpoint of the line geometry of the source segment is taken; if there is a starting point of another segment's line geometry that matches the endpoint, the second segment is a target segment for the source one. These pairs of segments can then be added to the `flows-to` relation. This is done for all segments in the VHA, after which the `flows-to` relation represents the complete system flow.

It is worth noting that not every segment has a follow-up segment.[2] For example, there are segments that end up in the sea, or in some special cases just stop. This does not always mean that a river stops at the end of that segment; the river can, for example, cross the Flemish border and subsequently not have any follow-up segment in the data set. Also, segments do not always have exactly one follow-up segment, since a river can split into two or more rivers that all flow on and, possibly, join again. In this situation, the endpoint of the segments will fall together with more than one starting point. Therefore, the `flows-to` relation can contain multiple tuples for a

specific segment, and this should be taken into account when devising algorithms for the search of flow paths, and also for the creation of the database itself.

We remark that, in general, the flows-to graph of a river system is acyclic, since naturally flowing water cannot flow from one location via some path to that same location (if the river system includes pumps, this might be different). Therefore, a `flows-to graph` is a directed acyclic graph.

It has been mentioned that the VHA data set is delivered as a shapefile where all segments and their properties are stored. In order to add the flow information, the file is loaded into a spatial relational database management system, namely PostgreSQL, equipped with the PostGIS extension (https://postgis.net). This table is denoted `wlas`. From it, the `flows-to table` is created as follows:

```
CREATE TABLE
flows _ to(source _ segment bigint, target _ segment bigint);
```

The new table can be filled using:

```
INSERT INTO flows _ to(source _ segment, target _ segment)
SELECT a.vhas, b.vhas
FROM wlas a, wlas b
WHERE ST _ StartPoint(b.geom) = ST _ EndPoint(a.geom);
```

This query cannot be directly executed on the VHA data set after it is imported into the PostgreSQL database. The reason for that is that the geometries in the VHA shapefile, and thus in the database, are multi-line geometries and the `ST _ StartPoint()` or the `ST _ EndPoint()` functions cannot take a multi-geometry as input. Therefore, the multi-line geometries must be converted to a single line geometry. The following statements create a new column `geomS` in the `wlas` table, with type line geometry defined using the map projection 31370 (which is the "Belgian Lambert 72" projection), and then convert each multi-line segment into a single line segment. After this pre-processing of the VHA data, the query above can be executed. We note that the usage of `b.geom` and `a.geom` needs to be replaced with the name of the newly created column, in this example `b.geomS` and `a.geomS`.

```
ALTER TABLE wlas
ADD COLUMN geomS geometry(LineString, 31370);
UPDATE wlas SET geomS = ST _ LineMerge(geom);
```

3.2.2 | Fixing errors

Some errors encountered during the creation of the data set need to be fixed, to obtain a usable database. These are discussed next.

Segments that do not match

Up until now two segments were defined as source and target segments if their ending and starting points coincide. However, if the difference between the two points is relatively small, they may still represent the same physical location (see Figure 2). To overcome such small mismatches, the comparison of the points needs to be relaxed, allowing a *tolerance* for considering two points to be the same. This can be addressed as follows for the original flows-to relation:

```
WHERE ST _ StartPoint(b.geom) = ST _ EndPoint(a.geom) OR
ST _ DWithin(ST _ StartPoint(b.geomS),ST _ EndPoint(a.geomS),1);
```

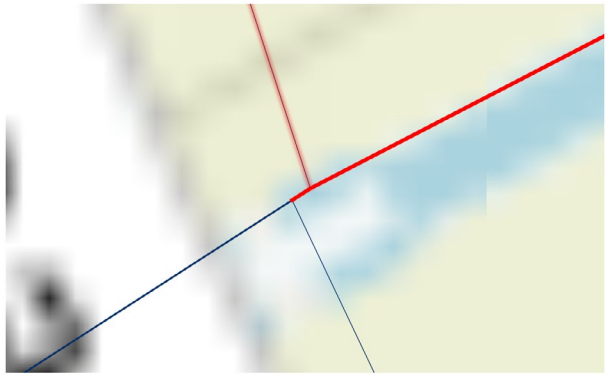

FIGURE 2 An example of a group of segments where the endpoint does not match exactly with other segments' start or end points

FIGURE 3 An example of two segments colliding in the VHA caused by an incorrect direction. The blue arrows indicate the direction encoded in the VHA

Here, the tolerance is set to 1 m (the Belgian Lambert 72 reference system implies meters). The value of the tolerance that is allowed depends on the problem. For the VHA data set, 1 m is adopted, based on empirical tests with different values. Using a tolerance also entails that there is a higher possibility of encountering *false positives*, meaning that two segments could appear to be matching although they are not. However, with the adopted value this is kept to a reasonably small probability.

Incorrect directions

In the real world, rivers often have a direction associated with them. A direction incorrectly encoded may lead to colliding segments or non-matching starting and ending points, as illustrated in Figure 3. The solution entails the following steps:

1. Execute the standard flows-to relation.
2. Find all unmatched segments and select the ones that have an incorrect-direction issue.
3. Store these segments in a temporary table, reverse the direction, and add the information back to the created relation.

Incomplete or unexpected data

Two further issues must be mentioned: isolated lines or segments; and border crossing and re-entering.

The first is rather unexpected in a data set that represents rivers or water streams. In the case of the VHA, it can happen that one line geometry or a small set of line geometries do not connect to the remainder of the segments in the data set. Those isolated segments form a static water body, like a small pond, where mainly runoff water is captured until it drains into the ground. By definition, the VHA only charts bodies of water or streams that have a flowing character. The segments, therefore, are unexpected data in the VHA data set. The influence of those cases is negligible on the overall data set because they only influence the results if a query, asking for a certain path (discussed in Section 5), starts in one such segment. Furthermore, this occurs with the smallest segments.

Border-crossing segments occur because there are administrative boundaries to a region which do not match to the natural boundaries existing in the landscape. The VHA data set contains the rivers, brooks and ditches of the Flanders region. However, the data set does not include the parts that are outside the region. The so-called border segments are the segments that leave the charted area and may re-enter further downstream. The border segments do not have any downstream segments in the `flows-to` relationship because such a segment does not exist, according to the assumptions made. The problem created by this gap at the border (the water that leaves the border segments ends up in another segment) is not captured by the present solutions. The distance between those segments can be a few meters but also a few kilometres. In this article it is assumed that the water that leaves the region does not re-enter in another stream. This entails that the information of the overall water flow in those cases is lost for good and cannot be captured in the final data set.

4 | STORING THE RIVER GRAPH IN RELATIONAL AND GRAPH DATABASES

In this section relational and graph databases are analysed with respect to their functional and modelling dimensions. First, since the analysis of rivers involves spatial data, the capability to handle these data is discussed first. Then the representation of the problem using the relational and graph data models is studied. Finally, typical recursive queries on the graph and relational database representations are discussed along two dimensions: intuitiveness and performance. In this sense, the questions to answer are: how easy and intuitive would it be to write typical queries, and how fast will they run? Query performance is studied in Sections 5 and 6.

4.1 | Handling spatial and non-spatial data

The data used in this article are mainly based on the VHA, which is distributed using shapefiles. Each version of the VHA is a new shapefile including the complete VHA and thus all geometries of the rivers in Flanders. Data are imported into the database and then converted to a suitable representation for querying.

Neither PostgreSQL nor Neo4j are geographical database management systems, which means that they do not have out-of-the-box support for geometric data like the VHA. However, this support can be added to them through extensions. In the case of PostgreSQL, PostGIS is the extension that adds support for geometric data, through a wide range of geometric functions (https://postgis.net/docs/PostGIS_Special_Functions_Index.html). PostGIS also provides functions allowing importation of a shapefile through the *shp2pgsql* functionality (https://postgis.net/docs/manual-1.4/ch04.html#id419979). At the time of writing, Neo4j releases do not provide functionalities to import shapefiles. As in PostgreSQL, the overall flow information must be created after the data are imported (although in PostgreSQL, PostGIS and pgRouting (http://pgrouting.org) provide topology creation functions to facilitates this process). There is also a software library that provides interaction with OpenStreetMap (OSM) (https://github.com/neo4j-contrib/osm) and includes a scalable importer which takes advantage of Neo4j spatial indices, and also provides some functions for routeing analysis. In addition, a new spatial library, mentioned above, contains algorithms for spatial analysis (https://github.com/neo4j-contrib/spatial-algorithms), although to a

much lesser extent, compared with the functionality provided by PostGIS. Finally, the APOC library, which comes with the current Neo4j versions (4.x at the time of writing) (https://github.com/neo4j-contrib/neo4j-apoc-procedures), contains functions for geodecoding on OSM (other map services can be configured). In summary, compared to PostgreSQL, Neo4j so far lacks a wide range of spatial functionality.

4.2 | Data model

The typical way of performing routeing or path-finding analysis would be to store data in relational databases, on which SQL queries could be run. The aim of these queries is to find paths in the network, aggregating data with respect to some dimensions (e.g., time, river category), or querying data with respect to some geographical feature, location, or POI. A problem with this approach, particularly with the huge volumes of data available nowadays, is the difference between the way in which data are modelled and stored (this was called "impedance mismatch" above). Given that the river topology can be considered a graph, storing river data using relations may seem unnatural, especially since current database technology provides solutions that allow graphs to be stored in native form, as mentioned in Section 2. Relational and graph data models also come with high-level query languages.

For the problem under study, rivers are modelled as a sequence of segments, connected to each other. This is the typical case of recursive relationships, extensively studied in database conceptual modelling. Dullea and Song (1999) give a taxonomy of this kind of relationships. The translation of recursive relationships to the relational model is straightforward and also well studied. Thus, following traditional database theory, the river system is represented as follows. There is a table to store the segments information, such as ID and properties:

```
wlas(vhas, name, ...)
```

The attribute `vhas` is used as the identifier of the segment, and called ID. There is also a table containing the binary relation `flowsto` is used, as discussed in Section 3.2.1, where for each segment the follow-up segments are stored:

```
flowsto(source, target).
```

The `source` and `target` columns contain the IDs of the segments (`vhas`).

In Neo4j, segments are represented as nodes, with label `:Segment` (and their corresponding properties), and the relation between the nodes is called `:flowsTo`, defined as follows: there is a relation `:flowsTo` from node A to node B if the water is able to flow to segment B from segment A.

We note that in both models, the reverse flow can be addressed when querying, therefore adding the inverse relation, namely :comesFrom, is not actually needed to indicate a flow from node B to node A.

4.3 | Expressing recursive queries on relational and graph databases

Typical queries required by the problem under study are of the form "Where can the water flow to?" (downstream query) and "Where does the water come from?" (upstream query). Based on these queries, other computations can be performed, such as height and speed of the flow, pollution spread models, and many more. We note that these are recursive queries, which are computationally expensive, since they often require computation of the transitive closure of the underlying graph, a well-known problem in database theory (see, for example, Bancilhon & Ramakrishnan, 1986; Li & Ross, 1993). Actually, the worst-case time complexity for computing the transitive closure of a directed graph is $O(n \cdot e)$, where n is the number of the nodes, and e is the number of the edges. The space complexity is $O(n^2)$. As an example, a classic algorithm is proposed by Schmitz (1983). It follows that this is

also a hard problem in graph databases. However, this article shows that the graph representation would better take advantage of the structure of the river system in order to query the database efficiently.

With the layout of the data defined in Section 4.2, the SQL downstream query from a starting segment, with ID `id _startsegment`, can be written as follows (the upstream query is analogous, and omitted due to space restrictions):

```
WITH RECURSIVE outcome(source, target) AS (
            (SELECT source, target
            FROM flowsto
            WHERE source = id _ startsegment)
        UNION
            SELECT flowsto.source, flowsto.target
            FROM outcome, flowsto
            WHERE flowsto.source = outcome.target)
    SELECT DISTINCT target FROM outcome;
```

Cypher, like SQL, is a high-level, declarative, programming language. It is specifically designed for graph structures, and is the language that comes with the Neo4j database. It uses nodes and relations as first-class citizens, although the output to a query can be a graph or a set of tuples. The Cypher query that computes the downstream query shown in SQL above reads:

```
MATCH (N:Segment)-[:flowsTo*]->(M:Segment)
WHERE N.vhas = id _ startsegment
RETURN DISTINCT M.vhas;
```

Note that both are recursive queries computing the transitive closure of the graph, and returning the nodes in the graph that can be reached starting from a given one. That is, the queries do not output the paths, just the reachable segments. In the case of SQL, listing the paths would be even more complex. For example, the query below computes the paths to each reachable segment:

```
WITH RECURSIVE outcome(source, target, path) AS (
            (SELECT flowsto.source, flowsto.target,
                ARRAY[flowsto.target]
            FROM flowsto
            WHERE flowsto.target = id _ startsegment)
        UNION
            SELECT flowsto.source, flowsto.target,
                outcome.path || Array[flowsto.target]
            FROM outcome, flowsto
            WHERE flowsto.target = outcome.source
                AND flowsto.target <> All(path))
     SELECT DISTINCT path FROM outcome;
```

In the case of Cypher, to compute and list the paths, it suffices to write:

```
MATCH path= (N:Segment {vhas:id _ startsegment})-
                        -[:flowsTo*]->(M:Segment)
RETURN DISTINCT path;
```

It can be seen that the structure of the Cypher query is far less complicated and more intuitive than its SQL counterpart, since it takes advantage of the graph structure. In this case, a basic `MATCH.. WHERE.. RETURN` structure suffices to express a recursive query. This is mainly because Cypher is developed as a query language for graphs and recursion is typical in these cases. The `(N:segment)-[:flowsTo*]->(M:segment)` pattern selects all nodes M that are reachable by following one or more edges in the graph, traversing the graph using the :flowsTo relation. In addition, the APOC library contains many functions that can be used to compute the query above in a more efficient way, using breadth-first and depth-first algorithms for expanding the nodes. An example of such a query is:

```
MATCH (n:Segment {vhas:6033614})
CALL apoc.path.expandConfig(n,
    {relationshipFilter: "flowsTo>", minLevel: 1})
    YIELD path AS path
RETURN path;
```

The expandConfig function expands the nodes of a graph, computing all the paths between a node and all the other ones in the graph. Moreover, most of the time, the structure of the river system is a tree (recall that the hydrological models introduced in Section 2 consider a river system as a tree rather than a graph). This allows the use of functions that compute the (directed) spanning tree of the starting node, which is even more efficient. This function expands a spanning tree reachable from the start node following a relationship up to a certain level adhering to the label filters indicated as arguments. The nodes returned collectively form a spanning tree. This is studied in detail in the next section.

5 | QUERYING THE RIVER DATABASE

This section discusses a collection of queries on the rivers database. The collection of queries was composed after consultation with several hydrologists. The queries are designed as a starting point for real-life challenges such as "Where does an observed pollution come from?" and "Where will the observed pollution go to? When will it arrive there?" These queries are then run on Neo4j and PostgreSQL, and the results reported in Section 6. The queries are expressed in Cypher and SQL, respectively. However, for the sake of space, only the former are shown here, since SQL is a well-known language, and the work is focused on graph databases. Nevertheless, the SQL queries are included in Appendix A. For clarity, the queries are organised into classes that account for their main characteristics. To allow an adequate comparison of Cypher and SQL queries, the required output formats are indicated for each query.

5.1 | Queries of type 1: Aggregation and similarity queries

The queries in this class are typical aggregation queries à la GraphOLAP (Gómez, Kuijpers, & Vaisman, 2020). Aggregations are performed over different properties used as categories and metrics. For example, Query 5.1 just uses the segment's length as a metric, while Query 5.2 aggregates this metric by segment category. Query 5.3 takes the length of the segments and compares them against the length of a given node, in order to obtain segments with similar lengths. The output formats are: for Query 5.1 a float number, for Query 5.2 a tuple of the form (key, length), and for Query 5.3 a list of (ID, length) pairs.

Query 5.1 *Compute the average segment length.*

```
MATCH (n:Segment)
RETURN  avg(n.lengte) AS avglength
```

Query 5.2 *Compute the average segment length by segment category.*

```
MATCH (n:Segment)
RETURN  n.catc as category, avg(n.lengte)
AS avglength order by category asc
```

Query 5.3 *Find all segments that have a length within a 10% margin of the length of segment with ID 6020612.*

```
MATCH (n:Segment {vhas:6020612})
WITH n.lengte as length
MATCH (m:Segment)
WHERE m.lengte < length*1.1 and m.lengte > length*0.9
RETURN m.vhas, m.lengte;
```

5.2 | Queries of type 2: Network typology

This class of queries addresses the computation of metrics of the river network configuration. Although the queries include aggregation (like those of Type 1), they are included in this class because of their main functional meaning.

Query 5.4 *For each segment find the number of incoming and outgoing segments.*

The output of this query is a set of tuples of the form (ID, #in, #out). The query reads in Cypher as follows:

```
MATCH (src:Segment)-[:flowsTo]->(n:Segment)-[:flowsTo]
      ->(target:Segment)
RETURN n.vhas as nodenbr, COUNT(DISTINCT src) as segIn,
       COUNT (DISTINCT target) as segOut
```

Query 5.5 *Find the segments with the maximum number of incoming segments.*

The output of this query is a list of segment IDs and an integer representing the maximum number of incoming segments.

```
MATCH (n:Segment)
OPTIONAL MATCH (src:Segment)-[:flowsTo]->(n)
WITH n, COUNT(distinct src) as indegree
WITH COLLECT ([n, indegree]) as tuples,
     MAX(indegree) as max
RETURN  [t in tuples WHERE t[1] = max |t]
```

The OPTIONAL statement works like a relational outer join. The COLLECT statement aggregates the results in a list of pairs, to which list comprehension functions are then applied. The elements in the list, with values equal to the maximum are returned.

Query 5.6 *Find the number of splits in the downstream path of segment 6020612.*

The output of this query is an integer number indicating the number of splits found.

```
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:flowsTo]->(r:Segment)
WITH p, count(DISTINCT r) as co WHERE co > 1
RETURN count(p)
```

Here, the spanningTree function from the APOC library is used. This function computes all simple paths that can be reached starting from a node in the graph, using breadth-first search by default. This is done visiting nodes only once. The `relationshipFilter` is "`flowsTo>`", indicating that the path must traverse only this relation, in the downstream direction. The function can be parameterised in many ways, for example, indicating the minimum and maximum levels in the path (here, the latter is omitted). A collection of paths is returned (`pp`), which is then flattened as a table with the UNWIND statement. All reachable nodes are obtained. For each node in this table, it is tested whether this node has more than one outgoing segments. If this is the case, there is a split. The node with `vhas`:6020612 is chosen for the test because it is one of the farthest from the sea, thus its flow downstream is one of the longest.

Query 5.7 *Find the number of in-flowing segments in the downstream path of segment 6020612.*

The output of this query is an integer giving the number of in-flowing segments found. An in-flowing segment is a segment that ends on the downstream path, but which is not a part of the path itself, that is, a segment that contributes to the flow of a given one.

```
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path AS pp
WITH  [p in NODES(pp) | p.vhas] as ids
UNWIND ids as id
WITH collect(DISTINCT id) as ids
MATCH (s:Segment)-[:flowsTo]->(p)
WHERE NOT s.vhas in ids AND p.vhas <> 6020612
      AND p.vhas in ids
RETURN count(DISTINCT s) as inflows
```

This query is similar to Query 5.6, also using the spanningTree function. List comprehension is used to obtain the node identifiers.

Query 5.8 *Determine if there is a loop in the downstream path of segment 6031518.*

Sometimes, when the level of the sea is higher than normal, the sea may get into the river flow and reverse its direction. Moreover, anthropogenic influences, such as barriers, dams and sluices, can create loops in the system.

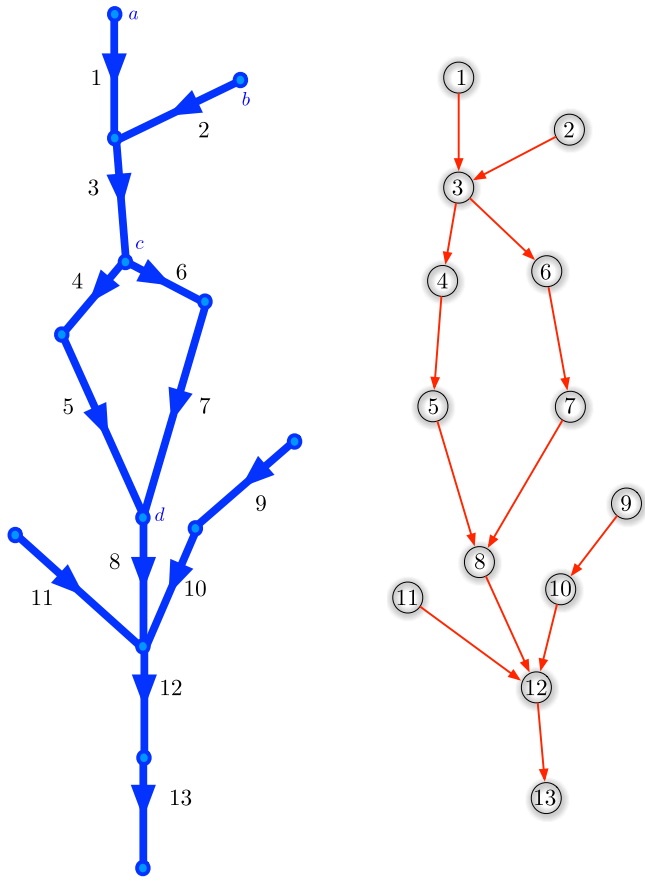

FIGURE 4 Nodes with more than one incoming flow

From a modelling point of view, in these cases, the graph will contain a cycle. This query finds out if this is the case in the graph under study. This also shows that, in order to obtain realistic modelling, the tree representation does not suffice, and a model like the one proposed in this article is needed. The output of the query is a Boolean.

```
MATCH (n:Segment {vhas:6031518})
CALL apoc.path.spanningTree(n, {relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path AS pp
WITH  [p in NODES(pp) | p] as nodelist
UNWIND nodelist as p
CALL apoc.path.expandConfig(p,
      {relationshipFilter:"flowsTo>", minLevel: 1,
      terminatorNodes:[p], whitelistNodes:nodelist})
yield path as loop
RETURN count(loop) >0 as loops
```

This query needs some explanation, which will also be used later. In this case, not only is the `spanningTree` function used, but also the `expandConfig` function. The left-hand side of Figure 4 shows the representation of the river as segments. Each edge represents a river segment, starting in one node and ending in another. The representation that was chosen for the graph is depicted on the right-hand side. Here, a segment becomes a node,

for example, the segment `c`, running from nodes 3 to 4, becomes the node `c`. It can be seen that segment `g`, for instance, receives flow from two incoming segments, namely `e` and `f`. If, for example, `a` is the starting segment, the `spanningTree` function would only capture one of the paths, the one which is first found by the algorithm. On the other hand, the `expandConfig` function finds all the paths. In a tree representation this problem would not arise, and the second CALL would not be needed, greatly simplifying the query. This function has a high computational cost, and should be used only if needed. For example, when the user only needs to obtain the nodes that can be reached from a given one, `spanningTree` should be used, since it is very efficient.

Expressiveness. The Cypher queries above are, in general, simpler than their SQL equivalent (shown in Appendix A), in particular for Queries 5.5–5.8. Writing the SQL code for the latter requires expert knowledge, while, even though the Cypher equivalents are not trivial, they basically require knowledge of the existence of the right functions. Further, when the `expandConfig` function is not required, the queries turn out to be very simple. In the case of SQL, the queries basically do not change for the two situations above.

5.3 | Queries of type 3: Path aggregation

The queries in this class aggregate a metric along a path. The length of a segment is used here, although for this scenario the average flow, or the average of any parameter reported by a sensor, could be used.

Query 5.9 *Find all paths downstream from the given start segment*.

There is no aggregate function in this query; the aggregation is given by the output, consisting of the IDs of the segments that can be reached from a given one, and the list of IDs of the corresponding paths.

```
PROFILE
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.expandConfig(n, {relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:flowsTo]->(r:Segment)
WITH r, count(DISTINCT p) as co WHERE co > 1
WITH collect(r) as pc
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.expandConfig(n,{relationshipFilter:
     "flowsTo>", minLevel:1,endNodes:pc}) YIELD path AS pp
WITH [p in NODES(pp) |p.vhas] AS nodelist
     WHERE size(nodelist) > 0
RETURN nodelist[size(nodelist)-1] as id, nodelist
UNION ALL
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:flowsTo]->(r:Segment)
WITH r, count(DISTINCT p) as co WHERE co = 1
WITH collect(r) as pc
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{relationshipFilter:
```

```
 "flowsTo>", minLevel:1,endNodes:pc}) YIELD path AS pp
WITH [p in NODES(pp)|p.vhas] AS nodelist
RETURN nodelist[size(nodelist)-1] as id, nodelist;
```

This query requires a trick, to make it possible to run in standard hardware. Since the expandConfig function is extremely costly, and the spanningTree function is very efficient for reachability, the former is only applied to compute the paths where there is more than one possible path for reaching a segments. This is computed in the upper subquery. The parameter `endNodes:pc` in the functions tells the algorithm to only expand the nodes in this list. The lower subquery uses the `spanningTree` function to compute the paths where there is only one way to reach the segment. The terms UNION and UNION ALL return the union of the results, without and with duplicates, respectively. This solution would probably not be efficient in a highly interconnected social network, since the `expandConfig` function computes all the paths between a node and all the other ones in the graph, which is computationally very expensive. On the other hand, the `spanningTree` function stops when it finds a path between the node being expanded and each other one. However, it is assumed that river networks are much less interconnected than a typical social network, and therefore it should work well, as shown in the experiments reported in Section 6.

Query 5.10 *Find the branches of downstream flow starting at a given position (identified by a segment's* vhas *ID), together with the length and number of segments of each branch.*

The output is a collection of tuples of the form (target segment ID, # of hops, length).

```
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{relationshipFilter:
   "flowsTo>", minLevel: 1}) YIELD path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:flowsTo]->(r:Segment)
WITH r, count(DISTINCT p) as co WHERE co = 1
WITH collect(r) as pc
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{}relationshipFilter:
   "flowsTo>", minLevel: 1,endNodes:pc}) YIELD path AS pp
WITH [p in NODES(pp) |p.vhas] AS nodelist,
reduce(longi= tofloat(0),n IN nodes(pp)|longi+n.lengte)
      AS segLen,
reduce(longi= 1,n IN nodes(pp)| longi + 1) AS nbrSeg
RETURN nodelist[size(nodelist)-1] as id, nbrSeg, segLen;
UNION
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n, {relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:flowsTo]->(r:Segment)
WITH r, count(DISTINCT p) as co WHERE co > 1
WITH collect(r) as pc
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.expandConfig(n,{relationshipFilter:
    "flowsTo>", minLevel:1,endNodes:pc}) YIELD path AS pp
```

```
WITH [p in NODES(pp) |p.vhas] AS nodelist,
reduce(longi = tofloat(0),n IN nodes(pp)|longi+n.lengte)
      AS segLen,
reduce(longi = 1,n IN nodes(pp)| longi + 1) AS nbrSeg
RETURN nodelist[size(nodelist)-1] as id, nbrSeg, segLen;
```

This is similar to the previous query, except for the aggregation of the lengths and number of segments. The reduce function computes the value resulting from the application of an expression on each successive element in a list, and accumulates these results as it proceeds. This allows the length of each branch (using, in this case, the property `lengte`) and the number of hops to be computed.

Query 5.11 *Find the length, the number of segments and the IDs of the segments, of the longest branch of upstream flow starting from a given segment*.

The output is a set of triples of the form (ID, length, # of segments). In this case the length is returned in meters.

```
PROFILE
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.expandConfig(n,{relationshipFilter:
     "<flowsTo", minLevel: 1}) YIELD path AS pp
WITH reduce(longi= tofloat(0), n IN nodes(pp)| longi
     + tofloat(n.lengte)) AS blength, Length(pp) as
     alength, [p in NODES(pp) |p.vhas] AS nodelist
WITH blength, alength, nodelist[size(nodelist)-1] as id
WITH id, max(blength) as ml,
     collect([id,blength,alength]) as coll
WITH id, ml, [p in coll WHERE p[0]= id
     AND p[1]=ml|p[2]] AS lhops
UNWIND lhops as hops
RETURN id,ml,hops order by id desc;
```

In this case, the upstream flow is requested. Therefore, the relationship filter now is "`<flowsTo`", indicating that the direction is reversed. This is why there is no need to specify and create the reversed `flowsto`, `comes-from`, relation in the graph. The tricky part in this query is to solve the cases where the longest physical branch is not the one with the maximum number of hops arriving to the same segment. The function `expandConfig` is used to compute all the alternative paths, and then reduce is used to compute the length of each branch. List comprehension is finally used to keep only the tuples that correspond to the branch of maximum length.

Query 5.12 *How many paths exist between two given segments X and Y?*

The output is an integer indicating the number of paths. This case is illustrated by the flow between segments c and g in Figure 3. To capture this case, again, the function expandConfig must be used.

```
MATCH (n:Segment {vhas:6020612}),
      (m:Segment {vhas: 7036554})
CALL apoc.path.expandConfig(n,
     {relationshipFilter:"<flowsTo", minLevel: 1,
     terminatorNodes:[m]}) yield path as pp
RETURN count(pp) as paths
```

Expressiveness. Queries in this class are quite complex to write, in both Cypher and SQL, except Query 5.12, which in Cypher only requires a function call. For the rest of the queries, complexity arises mainly from the situation depicted in Figure 4, which is a very particular case. Otherwise the queries become simpler (although not trivial, of course).

5.4 | Queries of Type 4: Conditions over paths

These queries only traverse certain branches of the rivers, indicated by conditions over properties of the paths or segments.

Query 5.13 *Find all branches starting at a given segment, reachable traversing the river Scheldt.*

The output is the ID of each final segment, and all the paths that lead to it.

```
PROFILE
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.expandConfig(n,{relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:flowsTo]->(r:Segment)
WITH r, count(DISTINCT p) as co
WHERE co > 1
WITH collect(r) as pc
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.expandConfig(n, {relationshipFilter:
     "flowsTo>", minLevel:1,endNodes:pc}) YIELD path AS pp
WITH [p in NODES(pp) WHERE p.strmgeb ="Schelde" |p.vhas]
     AS nodelist WHERE size(nodelist) > 0
RETURN nodelist[size(nodelist)-1] as id, nodelist
UNION ALL
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path AS pp
UNWIND NODES(pp) as p
MATCH (p)-[:flowsTo]->(r:Segment)
WITH r, count(DISTINCT p) as co
WHERE co = 1
WITH collect(r) as pc
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{relationshipFilter:
     "flowsTo>",minLevel: 1,endNodes:pc})
     YIELD path AS pp
WITH [p in NODES(pp) WHERE p.strmgeb ="Schelde"|p.vhas]
     AS nodelist
RETURN nodelist[size(nodelist)-1] as id, nodelist;
```

Since the query asks for all the paths, and not only for the segments, again the `spanningTree` function is not enough, and `expandConfig` must be used. The statement `[p in NODES(pp) WHERE p.strmgeb ="Schelde" |p.vhas]` keeps only the branches of the selected river. Experiments (not reported here, for the sake of space) have proven that this option is more efficient than including a parameter indicating a whitelist of the segments to be traversed.

Query 5.14 *List the length, the number of segments and the IDs of the segments of the branches starting from a given segment, that are part of the river Scheldt.*

The output are the triples (ID, length, # of segments) for each segment (only the shortest path information). This query is similar to Query 5.13, except for the final part. The computation of the paths is done analogously to the previous query. Thus, for the sake of space, only the final part is shown.

```
MATCH (n:Segment {vhas:'6020612'})
CALL apoc.path.spanningTree(n,{relationshipFilter:
     "flowsTo>", minLevel: 1}) YIELD path as pp
     .....
WITH [p in NODES(pp) WHERE p.strmgeb ="Schelde" |p.vhas]
    AS nodelist, reduce(longi= tofloat(0),n IN nodes(pp)|
    CASE WHEN n.strmgeb ="Schelde" THEN longi + n.lengte
    ELSE longi END) AS length, reduce(longi= 1,n
    IN nodes(pp)|CASE WHEN n.strmgeb ="Schelde"
    THEN longi + 1 ELSE longi END) AS segCount
RETURN nodelist[size(nodelist)-1] as id, segCount, length
```

The reduce statements compute the lengths of the segments and the number of segments in each path. The statement `CASE WHEN n.strmgeb ="Schelde" THEN longi +tofloat(n.lengte) ELSE longi END` is used to aggregate only the requested branches in the reduce statement. We note that this solution captures all the alternative paths when there is more than one way of reaching a certain node.

Expressiveness. Queries in this class are, as can be seen, very complex. Query 5.13 in SQL is much simpler, but Query 5.14 requires a deep knowledge of SQL programming.

5.5 | Queries of Type 5: Spatial queries

Finally, a class of queries including spatial data are proposed.

Query 5.15 *Find all segments reachable from the segment closest to Antwerp's Groenplaats.*[3]

The output is a list of segment IDs; no path information is required.

```
CALL apoc.spatial.geocodeOnce('Groenplaats
    Antwerpen Flanders Belgium')
     YIELD location as ini
MATCH (n:Segment)
WITH n, ini,distance(
  point({longitude:n.source _ long, latitude:n.source _ lat}),
  point({longitude:ini.longitude, latitude:ini.latitude})
   ) as d
```

```
WITH n, d order by d asc limit 1
CALL apoc.path.spanningTree(n,
     {relationshipFilter:"flowsTo>", minLevel: 1})
     YIELD path as pp
UNWIND NODES(pp) as p
RETURN p.vhas;
```

Here, the APOC function geocodeOnce is used to find the starting point, from which the reachable segments are computed. Antwerp's Groenplaats is taken as the reference. Then, Cypher's built-in distance function computes the distance between Groenplaats and the closest river segment. The rest, is analogous to the previous queries.

Query 5.16 *Find the segments that belong to the downstream path and that are at most 3 km from the start segment, together with the minimum distance from the start to the segment.*

The output is a list of segment IDs, and the length of the shortest path, in meters. Since the minimum distance is required, again the expandConfig function must be used. Only the portion of the query related to the computation of the distance is shown, the rest is analogous to the previous queries.

```
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n, {relationshipFilter:
     "flowsTo>", minLevel: 1})
      YIELD path AS pp
...
...
CALL apoc.path.expandConfig(n, {relationshipFilter:
   "flowsTo>", minLevel:1,endNodes:pc}) YIELD path AS pp
UNWIND NODES(pp) AS p
WITH distance(point({longitude:n.source _ long,
  latitude:n.source _ lat}), point({longitude:p.source _ long,
  latitude:p.source _ lat})) as dist, p WHERE dist < 3000
RETURN DISTINCT p.vhas, min(dist)
UNION
...
...
MATCH (n:Segment {vhas:6020612})
CALL apoc.path.spanningTree(n,{relationshipFilter:
  "flowsTo>", minLevel: 1,endNodes:pc}) YIELD path AS pp
UNWIND NODES(pp) AS p
WITH distance(point({longitude:n.source _ long,
  latitude: n.source _ lat}), point({longitude:
  p.source _ long, latitude: p.source _ lat}))
  as dist, p WHERE dist < 3000
RETURN DISTINCT p.vhas, min(dist);
```

In this case, the distance function is used to compute which segments are less than 3 km from the starting point.

Expressiveness. Here, comparing the Cypher queries with the SQL and PostGIS queries in Appendix A, it appears clear that the degree of maturity of spatial capabilities of PostGIS gives SQL a clear edge over the graph alternative. Spatial support is still needed in graph databases.

6 | EXPERIMENTAL EVALUATION

The queries in Section 5 are run on the Neo4j database which is designed and populated as described in Section 4.2. Furthermore, in order to compare performance against the relational alternative, the queries are written in the SQL language, and executed on a PostgreSQL database. For the sake of fairness of comparison, the type of output, as well as the results of the SQL queries, are the same as the ones corresponding to the Cypher queries in Section 5. Both databases are fully indexed in order to obtain the best possible query performance. Indices are created over all attributes that are mentioned in the queries (the segment identifiers, `strmgeb`, `lengte`, `catc`, etc.). Neo4j provides two classes of indices: native B-tree and full-text search indices. In this work, native B-tree indices are used. Figure 5 shows the index configuration used for Neo4j.

In PostgreSQL, the tables and indices are stores in the same tablespace. The index type is the default B-tree for all indices. For example, for the source attribute in the `wlas` and `flowsto` tables:

```
CREATE INDEX edgesour
    ON public.wlas USING btree
    (source ASC NULLS LAST)
    TABLESPACE pg_default;
CREATE INDEX flowsto_source_idx
    ON public.flowsto USING btree
    (source ASC NULLS LAST)
    TABLESPACE pg_default;
```

The specific segments used in the queries are chosen based on the following criteria. For downstream flows, the starting segment is chosen close to the start of the flow. For queries analysing upstream flow, starting segments close to the end of the flow are chosen. Although several segments were considered as candidates, only a representative one is reported in this work.

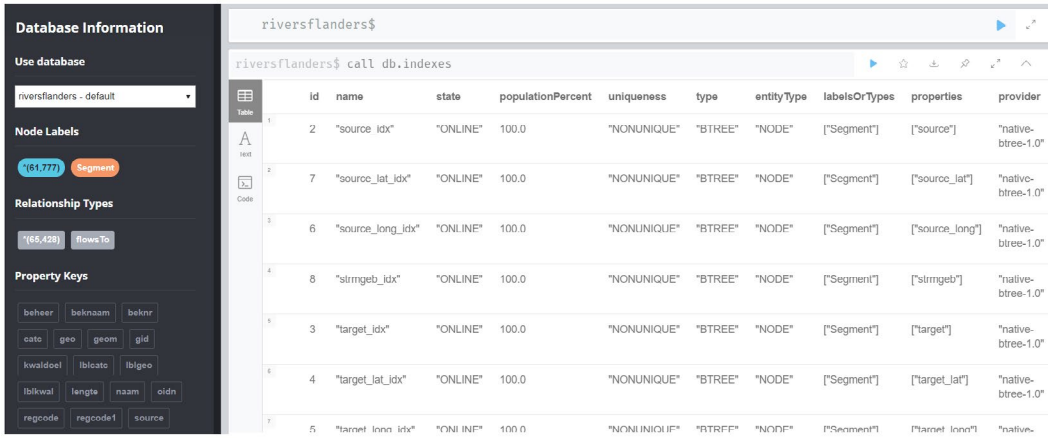

FIGURE 5 Neo4j index configuration

6.1 | Experiments setup

For the Neo4j database, the numbers of nodes and edges are given in Table 1. For the PostgreSQL database, the table from which the edges age obtained, called : `wlas`, has 61,777 tuples, and the `flowsto` table, containing overall flow information, 65,428 tuples. The queries are run on a machine with an i7 7700 processor at 2.8 GHz, with 32 GB of RAM and a 1 TB disk. The execution times reported are the averages of five runs of each experiment.

6.2 | Discussion

Table 2 summarises the test results. The rightmost column gives the ratio between the execution times on Neo4j and PostgreSQL. The best execution times for each query have been highlighted in bold. When the value is set to ∞ this means that the query ran for more than 10 min without finishing. The discussion that follows is organised according to the query classes defined in Section 5.

TABLE 1 Number of nodes and edges in the Rivers graph database

Type	Name	Size (#)
Node	`Segment`	61,777
Edge	`flowsto`	65,428
Total	# Objects	126,205

TABLE 2 Execution times for the example queries

		(3)	(4)	
Type of Query	**Query**	**Neo4j (ms)**	**PostgreSQL (ms)**	**(3)/(4)**
Aggregation & similarity	5.1	**79**	94	0.84
Aggregation & similarity	5.2	111	**103**	1.07
Aggregation & similarity	5.3	**96**	116	0.83
Network Topology & pattern	5.4	**14**	258	0.05
Network Topology & pattern	5.5	**184**	193	0.95
Network Topology & pattern	5.6	**35**	51	0.69
Network Topology & pattern	5.7	319	**47**	6.79
Network Topology & pattern	5.8	**663**	2,200	0.30
Path Aggregation & pattern	5.9	**1,740**	∞	N/A
Path Aggregation & pattern	5.10	**1,820**	∞	N/A
Path Aggregation & pattern	5.11	**711**	47,000	0.015
Path Aggregation & pattern	5.12	**1**	47	0.02
Conditions over paths	5.13	**1,914**	11,300	0.17
Conditions over paths	5.14	**1,596**	∞	N/A
Spatial	5.15	**388**	613	0.55
Spatial	5.16	26,038	**48**	542

Note: The fastest execution times are shown in bold numbers.

The results show that almost all queries run much faster in Neo4j. Although these results could be expected for transitive-closure-like queries, surprisingly, queries of type 1 (aggregate queries) written in Cypher also outperformed SQL queries, except for Query 5.2. For topological queries (type 2), Cypher clearly outperforms SQL, except for two of the queries. Likewise, performance is, in some cases, orders of magnitude better in Cypher for queries of types 3 and 4 (path queries), which encode the computation of the reachability in the graph and conditions and/or aggregations over the paths. Also surprising is the result for queries of type 5, where spatial functions are used. In this case, however, it is necessary to point out that spatial capabilities of Neo4j are not even close to those of PotGIS, as already commented. Nevertheless, the results are quite good (although, of course, far from conclusive). Also in this case, we note that in Query 5.15 the coordinates are computed with the built-in OSM service whereas in PostgreSQL they are hard-coded into the query, and even in this case, performance in better for Neo4j.

Another point that is worth a discussion, is the comparison between using the `spanningTree` function to compute the nodes reachable from a given one, against the simple Cypher's built-in transitive closure computation (the "*" function). The latter is orders of magnitude worse. However, the `expandConfig` function, which is needed when all the paths must be returned, and not just the nodes reachable from a certain one, is not as efficient, since it computes all the paths in the transitive closure.

Also, the analysis of the queries in Section 5 suggests that, in general, expressing queries in a graph-based high-level language results in simpler, more concise, and more intuitive expressions than their SQL equivalents. However, there are situations, typical in NoSQL databases, where the way in which a query is written affects the performance. This particularly occurs when all the paths must be computed, and the river system is modelled as a graph. When only a segment's reachability is required, or the river system can be modelled as a tree, or alternative paths are not needed, the Cypher expressions can be highly simplified, while SQL queries still require the transitive closure of the relation to be computed.

7 | CONCLUSION AND FUTURE WORK

This article uses a real-world case, based on the Flemish river system, to study the plausibility of using graph databases to represent, store and query river data. It also presents a traditional relational database implementation, and compares both alternatives. The data preparation tasks are described, as well as the data models used. Finally, a collection of queries are defined and executed on PostgreSQL and Neo4j databases, expressed in SQL and in Cypher, the high-level query languages for each respective database. The queries are run and the results discussed and reported.

The study suggests that river systems, and other kinds of transportation networks, can be modelled as graphs and implemented using graph databases, on which queries are, in general, more easily expressed using high-level graph query languages, in this particular case, Cypher. The results also show that queries involving path computation run faster overall on graph databases, since their underlying data structures are designed to achieve fast path traversal. In contrast, a relational representation requires writing recursive queries to compute the transitive closure of the graph, which affects query efficiency, since the relational representation does not capture the graph structure appropriately, a problem known in database modelling as "impedance mismatch". Five types of queries were studied, including aggregate, path, and spatial queries. Only three out of 16 queries delivered better performance in the relational version. In particular, in path computation, where the graph representation is crucial, the difference reaches orders of magnitude in favour of Cypher. However, it is worth noting that these results were obtained with the algorithms provided in Neo4j libraries, not with Cypher's built-in transitive closure computation. Nevertheless, long path traversals like the ones required in this problem are clearly not appropriately handled by the relational model, since they require multiple self-joins of the table containing the relationships between the river segments. It must be mentioned that intensive, advanced SQL query tuning was outside the scope of this

work. Rather the intention was to investigate the feasibility of using graph databases to model river networks. In summary, the results obtained in this work suggest that graph databases can be a good alternative for analysing large volumes of river data, like those in the IoW project.

Future work is mainly oriented towards scaling this problem for larger volumes, for which parallel processing may be needed. There are many parallel processing graph databases (e.g., GraphFrames (https://graphframes.github.io/graphframes/docs/_site/index.html), Janusgraph (http://janusgraph.org/)) that may take advantage of the characteristics of graphs like the ones studied here. Even Neo4j has recently presented a scalable version in the cloud. Other future work involves a generalisation to other transportation networks such as road, computer, sewage and heat networks.

ACKNOWLEDGEMENTS

Erik Bollen was supported by the Bijzonder Onderzoeksfonds (BOF) from UHasselt with reference BOF20OWB27 and by VITO with project reference 2010478. Alejandro Vaisman was partially supported by Project PICT 2017-1054 from the Argentinian Scientific Agency.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available in the "Vlaamse Hydrologische Atlas" at http://www.geopunt.be/catalogus/datasetfolder/020a452d-8cd2-41b7-9c64-2be367668837. These data were derived from the following resources available in the public domain: https://www.geopunt.be/catalogus/datasetfolder/020a452d-8cd2-41b7-9c64-2be367668837.

ORCID

Erik Bollen https://orcid.org/0000-0002-9287-1094
Rik Hendrix https://orcid.org/0000-0002-1572-1279
Bart Kuijpers https://orcid.org/0000-0001-5774-0948
Alejandro Vaisman https://orcid.org/0000-0002-3945-4187

ENDNOTES

[1] Graph databases are called "native" if they use specialised data structures for storing data. If, however, they provide an interface for other kinds of storage (e.g., relational databases), they are called "non-native".

[2] For a given source segment, the corresponding targets are considered to be follow-up segments.

[3] Groenplaats is the main square in the city of Antwerp.

REFERENCES

Ahani, A., Shourian, M., & Rahimi Rad, P. (2018). Performance assessment of the linear, nonlinear and nonparametric data driven models in river flow forecasting. *Water Resources Management*, *32*(2), 383–399. https://doi.org/10.1007/s11269-017-1792-5

Angles, R. (2012). A comparison of current graph database models. In *Proceedings of ICDE Workshops*, Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA (pp. 171–177). Piscataway, NJ: IEEE. https://doi.org/10.1109/ICDEW.2012.31

Angles, R. (2018). The property graph database model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, Cali, Colombia. Retrieved from http://ceur-ws.org/Vol-2100/paper26.pdf

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., & Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys*, *50*(5), 68:1–68:40. https://dl.acm.org/doi/10.1145/3104031

Angles, R., & Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, *40*(1), 1:1–1:39. https://doi.org/10.1145/1322432.1322433

Bancilhon, F., & Ramakrishnan, R. (1986). An amateur's introduction to recursive query processing strategies. In C. Zaniolo (Ed.), *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, Washington, DC (pp. 16–52). New York, NY: ACM Press.

Batra, S., & Tyagi, C. (2012). Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering*, *2*(2), 509–512. Retrieved from https://www.ijsce.org/wp-content/uploads/papers/v2i2/B0625042212.pdf

Brouwers, J., Peeters, B., Van Steertegem, M., & Van Lipzig, N. (2015). *MIRA Climate Report 2015*. Technical report. VMM, Aalst, Belgium.

Chen, C., Yan, X., Zhu, F., Han, J., & Yu, P. S. (2009). Graph OLAP: A multi-dimensional framework for graph data analysis. *Knowledge and Information Systems*, *21*(1), 41–63. https://doi.org/10.1007/s10115-009-0228-9

da Silva, M. C. T., Times, V., & Renso, C. (2015). SWOT: A conceptual data warehouse model for semantic trajectories. In *Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP, DOLAP 2015*, Melbourne, VIC, Australia (pp. 11–14). New York, NY: ACM.

Daltio, J., & Medeiros, C. B. (2015). Hydrograph: Exploring geographic data in graph databases. *Revista Brasileira de Cartografia*, *68*(6). Retrieved from http://www.seer.ufu.br/index.php/revistabrasileiracartografia/article/view/44491

Demir, I., & Szczepanek, R. (2017). Optimization of river network representation data models for web-based systems. *Earth and Space Science*, *4*(6), 336–347. https://doi.org/10.1002/2016EA000224

Dullea, J., & Song, I. (1999). A taxonomy of recursive relationships and their structural validity in ER modeling. In *Conceptual Modeling—ER '99*, *18th International Conference on Conceptual Modeling*, *Paris*, *France*, *November*, *15–18*, *1999*, *Proceedings* (Lecture Notes in Computer Science, Vol. *1728*, pp. 384–398). Berlin, Germany: Springer.

Fileto, R., May, C., Renso, C., Pelekis, N., Klein, D., & Theodoridis, Y. (2015). The Baquara2 knowledge-based framework for semantic enrichment and analysis of movement data. *Data & Knowledge Engineering*, *98*, 104–122. https://doi.org/10.1016/j.datak.2015.07.010

Gobin, A. (2012). Impact of heat and drought stress on arable crop production in Belgium. *Natural Hazards and Earth System Science*, *12*(6), 1911–1922. https://doi.org/10.5194/nhess-12-1911-2012

Gómez, L. I., Kuijpers, B., & Vaisman, A. A. (2019). Analytical queries on semantic trajectories using graph databases. *Transactions in GIS*, *23*(5), 1078–1101. https://doi.org/10.1111/tgis.12556

Gómez, L. I., Kuijpers, B., & Vaisman, A. A. (2020). Online analytical processing on graph data. *Intelligent Data Analysis*, *24*(3), 515–541. https://doi.org/10.3233/IDA-194576

Hartig, O. (2014). *Reconciliation of RDF* and property graphs*. Preprint, arXiv:1409.3288.

Heuvelmans, G., Muys, B., & Feyen, J. (2004). *Analysis of the spatial variation in the parameters of the SWAT model with application in Flanders, Northern Belgium*. Technical Report 5.

Li, Z., & Ross, K. (1993). *On the cost of transitive closures in relational databases*. Technical Report CUCS-004-93. Columbia University.

Makris, A., Tserpes, K., Anagnostopoulos, D., Nikolaidou, M., & de Macedo, J. A. F. (2019). Database system comparison based on spatiotemporal functionality. In B. C. Desai, D. Anagnostopoulos, Y. Manolopoulos, & M. Nikolaidou (Eds.), *Proceedings of the 23rd International Database Applications & Engineering Symposium*, *IDEAS 2019*, Athens, Greece (pp. 21:1–21:7). New York, NY: ACM.

Parent, C., Spaccapietra, S., Renso, C., Andrienko, G., Andrienko, N., Bogorny, V., … Yan, Z. (2013). Semantic trajectories modeling and analysis. *ACM Computing Surveys*, *45*(4), 42:1–42:32. https://doi.org/10.1145/2501654.2501656

Robinson, I., Webber, J., & Eifrém, E. (2013). *Graph databases*. Sebastopol, CA: O'Reilly Media.

Ruback, L., Casanova, M. A., Raffaetà, A., Renso, C., & Vidal, V. (2016). Enriching mobility data with linked open data. In *Proceedings of the 20th International Database Engineering and Applications Symposium*, *IDEAS 2016*, Montreal, QC, Canada (pp. 173–182). New York, NY: ACM.

Schmitz, L. (1983). An improved transitive closure algorithm. *Computing*, *30*(4), 359–371. https://doi.org/10.1007/BF02242140

Solomatine, D. P., & Ostfeld, A. (2008). Data-driven modelling: Some past experiences and new approaches. *Journal of Hydroinformatics*, *10*(1), 3–22. https://doi.org/10.2166/hydro.2008.015

Vicknair, C., Macia, C., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010). A comparison of a graph database and a relational database: A data provenance perspective. In *ACM SE '10: Proceedings of the 48th Annual Southeast Regional Conference* (pp. 42:1–42:6). New York, NY: ACM.

Zhao, P., Li, X., Xin, D., & Han, J. (2011). Graph Cube: On warehousing and OLAP multidimensional networks. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, Athens, Greece (pp. 853–864). New York, NY: ACM.

APPENDIX A.

SQL QUERIES

Query 5.1 *Compute the average segment length.*

```
SELECT avg(lengte) AS avglength
FROM edges;
```

Query 5.2 *Compute the average segment length by segment category.*

```
SELECT catc AS category, avg(lengte) AS avglength
FROM  wlas
GROUP BY catc;
```

Query 5.3 *Find all segments that have a length within a 10% margin of the length of segment with ID 6020612.*

```
SELECT vhas, lengte
FROM wlas
WHERE lengte <= 1.1*
   (SELECT lengte from edges WHERE vhas=6020612)
   AND lengte >= 0.9*(SELECT lengte
   FROM edges
   WHERE vhas=6020612)
```

Query 5.4 *For each segment find the number of incoming and outgoing segments.*

```
SELECT segments.vhas, count(DISTINCT flowstoB.source) AS segIn,
       count(DISTINCT flowstoA.target) AS segOut
FROM wlas as segments, flowsto as
     flowstoA, flowsto as flowstoB
WHERE segments.vhas = flowstoA.source
      AND segments.vhas = flowstoB.target
GROUP BY segments.vhas;
```

Query 5.5 *Find the segments with the maximum number of incoming segments.*

```
SELECT target as vhas, segIn FROM
    (SELECT target, count(flowsto.source) AS segIn
    FROM flowsto
    GROUP BY flowsto.target) AS myTable
WHERE segIn = (SELECT max(segIn) FROM
  (SELECT flowsto.target,
          count(flowsto.source) AS segIn
   FROM flowsto
   GROUP BY flowsto.target) AS tt);
```

Query 5.6 *Find the number of splits in the downstream path of segment 6020612.*

```
SELECT count(source) FROM
(WITH RECURSIVE outcome(source, target) AS (
        (SELECT source, target
        FROM flowsto
        WHERE source = 6020612)
        UNION
        SELECT flowsto.source, flowsto.target
        FROM outcome, flowsto
        WHERE flowsto.source = outcome.target )
    SELECT source, count(target) AS segOut
    FROM outcome
    GROUP BY source) AS myTable
WHERE segOut > 1;
```

Query 5.7 *Find the number of in-flowing segments in the downstream path of segment 6020612.*

```
SELECT sum(diff)
FROM (
  SELECT myTable.target, count(source)-segIn as diff
  FROM
    (WITH RECURSIVE outcome(source, target) AS (
        (SELECT source, target
        FROM flowsto
        WHERE source = 6020612)
        UNION
        SELECT flowsto.source, flowsto.target
        FROM outcome, flowsto
        WHERE flowsto.source = outcome.target )
    SELECT target, count(source) AS segIn
    FROM outcome
    GROUP BY target) AS myTable, flowsto
    WHERE myTable.target = flowsto.target
    GROUP BY myTable.target, segIn) AS secTable;
```

Query 5.8 *Determine if there is a loop in the downstream path of segment 6031518.*

```
WITH RECURSIVE outcome(source, target, again, path) AS (
   (SELECT source, target, 0, ARRAY[source]
    FROM flowsto
    WHERE source = 6031518)
    UNION
    SELECT flowsto.source, flowsto.target,
    CASE WHEN flowsto.source <> All(path) THEN 0
         ELSE 1 END, outcome.path||Array[flowsto.source]
```

```
    FROM outcome, flowsto
    WHERE flowsto.source = outcome.target AND
          outcome.again <> 1)
    SELECT count(source)>0 FROM outcome where again=1;
```

Query 5.9 *Find all paths downstream from the given start segment.*

```
WITH RECURSIVE outcome(source, target, path) AS (
    (SELECT flowsto.source, flowsto.target,
            ARRAY[flowsto.source]
    FROM flowsto
    WHERE flowsto.source = 6020612)
    UNION
    SELECT flowsto.source, flowsto.target, outcome.path
           ||Array[flowsto.source]
    FROM outcome, flowsto, wlas
    WHERE flowsto.source = outcome.target AND
          flowsto.source <> All(path))
    SELECT json _ agg(array _ to _ json(outcome.path)) AS paths
    FROM outcome
    WHERE  0=(SELECT count(target)
              FROM outcome as cin
              WHERE outcome.target=cin.source)
    GROUP BY outcome.target;
```

Query 5.10 *Find the branches of downstream flow starting at a given position (identified by a segment's vhas ID), together with the length and number of segments of each branch.*

```
WITH RECURSIVE outcome(source, target, path, length, segCount)
AS (
  SELECT flowsto.source, flowsto.target, ARRAY[flowsto.source],
  w1.lengte + w2.lengte, 1
  FROM flowsto, wlas as w1, wlas as w2
  WHERE flowsto.source = 6020612 and flowsto.source = w1.vhas
        and flowsto.target = w2.vhas
  UNION
  SELECT flowsto.source, flowsto.target, outcome.path
        || Array[flowsto.source], outcome.length + wlas.lengte,
        outcome.segCount + 1
  FROM outcome, flowsto, wlas
  WHERE flowsto.source = outcome.target AND
        wlas.vhas = flowsto.target AND flowsto.source <> All(path))
  SELECT target, path, length, segCount FROM outcome
  WHERE 0=(SELECT count(target) FROM outcome as cin
           WHERE outcome.target=cin.source);
```

Query 5.11 *Find the length, the number of segments and the IDs of the segments, of the longest branch of upstream flow starting from a given segment.*

```
WITH RECURSIVE outcome(source, target, path,
               length, segCount) AS (
  (SELECT flowsto.source, flowsto.target,
         ARRAY[flowsto.target], w1.lengte
          + w2.lengte, 1
   FROM flowsto, wlas as w1, wlas as w2
   WHERE flowsto.target = 6020612 AND
         flowsto.source = w1.vhas AND
         flowsto.target = w2.vhas)
   UNION
   SELECT flowsto.source, flowsto.target,
        outcome.path || Array[flowsto.target],
        outcome.length + wlas.lengte,
        outcome.segCount + 1
   FROM outcome, flowsto, wlas
   WHERE flowsto.target = outcome.source
         AND wlas.vhas = flowsto.source
         AND flowsto.target <> All(path))
SELECT source, min(length), min(segCount)
FROM outcome
GROUP BY outcome.source;
```

Query 5.12 *How many paths are there between two given segments X and Y?*

```
WITH RECURSIVE outcome(source, target, path) AS (
 (SELECT flowsto.source, flowsto.target,
         ARRAY[flowsto.target]
  FROM flowsto
  WHERE flowsto.target = 6020612)
  UNION
  SELECT flowsto.source, flowsto.target, outcome.path
         || Array[flowsto.target]
  FROM outcome, flowsto
  WHERE flowsto.target = outcome.source AND
  flowsto.target <> All(path) AND 7036554 <> All(path))
SELECT count(DISTINCT path)
FROM outcome
WHERE 7036554 = Any(path);
```

Query 5.13 *Find all branches starting at a given segment, reachable traversing the river Scheldt.*

```
WITH RECURSIVE outcome(source, target, path) AS (
 (SELECT flowsto.source, flowsto.target,
         ARRAY[flowsto.source]
```

```
 FROM flowsto
 WHERE flowsto.source = 6020612)
 UNION
 SELECT flowsto.source, flowsto.target, outcome.path
        || Array[flowsto.source]
 FROM outcome, flowsto, wlas
 WHERE flowsto.source = outcome.target AND flowsto.source
       <> All(path) AND wlas.vhas = flowsto.source
       AND strmgeb = 'Schelde')
SELECT outcome.target, json _ agg(array _ to _ json(path))
FROM outcome
GROUP BY outcome.target;
```

Query 5.14 *List the length, the number of segments and the IDs of the segments of the branches starting from a given segment, that are part of the river Scheldt.*

```
WITH RECURSIVE outcome(source, target, path,
     length, segCount) AS (
 (SELECT flowsto.source, flowsto.target,
         ARRAY[flowsto.source], w1.lengte + w2.lengte,1
  FROM flowsto, wlas as w1, wlas as w2
  WHERE flowsto.source = 6020612 and flowsto.source =
        w1.vhas and flowsto.target = w2.vhas)
  UNION
  SELECT flowsto.source, flowsto.target,
         outcome.path || Array[flowsto.source],
         outcome.length + wlas.lengte, outcome.segCount+1
  FROM outcome, flowsto, wlas
  WHERE flowsto.source = outcome.target AND
        flowsto.source <> All(path) AND wlas.vhas =
        flowsto.target AND strmgeb = 'Schelde')
  SELECT DISTINCT outA.target, outA.length, outB.segCount
  FROM outcome as outA, outcome as outB
  WHERE outA.target = outB.target AND
        outA.length=(SELECT min(length)
  FROM outcome as c2 WHERE c2.target=outA.target)
       AND outB.segCount= (SELECT min(segCount)
                           FROM outcome as c3
                           WHERE c3.target=outB.target);
```

Query 5.15 *Find all segments reachable from the segment closest to Antwerp's Groenplaats.*

```
WITH RECURSIVE outcome(vhas) AS (
   (SELECT wlas.vhas
    FROM wlas
    ORDER BY ST _ Distance(ST _ Point(source _ long, source _ lat),
    ST _ Point(4.4016, 51.2192)) LIMIT 1)
```

```
   -- 51.2192, 4.4016 are coordinates of Groenplaats Antwerpen
   UNION
   SELECT flowsto.target
   FROM outcome, flowsto
   WHERE outcome.vhas = flowsto.source)
SELECT DISTINCT vhas FROM outcome;
```

Query 5.16 *Find the segments that belong to the downstream path and that are at most 3 km of the start segment, together with the minimum distance from the start to the segment.*

```
WITH RECURSIVE outcome(vhas, path, dist, geom) AS (
   (SELECT wlas.vhas, ARRAY[vhas], 0.0::double precision, geom
    FROM   wlas
    WHERE vhas = 6020612)
    UNION ALL
    SELECT flowsto.target, outcome.path || Array[flowsto.target],
    ST _ Distance(ST _ StartPoint(ST _ LineMerge(wlas.geom)),
    ST _ StartPoint(ST _ LineMerge(outcome.geom))), outcome.geom
    FROM outcome, flowsto, wlas
    WHERE outcome.vhas = flowsto.source AND
          flowsto.target = wlas.vhas
    AND flowsto.target <> All(path) AND
        ST _ Distance(ST _ StartPoint(ST _ LineMerge(wlas.geom)),
        ST _ StartPoint(ST _ LineMerge(outcome.geom))) < 3000)
SELECT vhas, dist FROM outcome;
```