Unified Spatial Analytics from Heterogeneous Sources with Amazon Redshift

Nemanja Borić
nemanjab@amazon.com

Hinnerk Gildhoff
hinnerk@amazon.com

Menelaos Karavelas
karavem@amazon.com

Ippokratis Pandis
ippo@amazon.com

Ioanna Tsalouchidou
ioantsa@amazon.com

ABSTRACT

Enterprise companies use spatial data for decision optimization and gain new insights regarding the locality of their business and services. Industries rely on efficiently combining spatial and business data from different sources, such as data warehouses, geospatial information systems, transactional systems, and data lakes, where spatial data can be found in structured or unstructured form. In this demonstration we present the spatial functionality of Amazon Redshift and its integration with other Amazon services, such as Amazon Aurora PostgreSQL and Amazon S3. We focus on the design and functionality of the feature, including the extensions in Redshift's state-of-the-art optimizer to push spatial processing close to where the data is stored.

ACM Reference Format:
Nemanja Borić, Hinnerk Gildhoff, Menelaos Karavelas, Ippokratis Pandis, and Ioanna Tsalouchidou. 2020. Unified Spatial Analytics from Heterogeneous Sources with Amazon Redshift. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384704

1 INTRODUCTION

Spatial data is one dimension used in data warehouses to analyse business data. Enterprise companies use it to answer the *where*-question. *Where* are my best and worst areas for selling my products? *Where* are anomalies in my network? *Where* do I need to start a new location for my business? These are just some examples, but they already show that spatial analysis can be applied to a wide variety of industries. Nearly all data has a geo-reference and spatial data allows us to get new insights. The challenge is that not all data is stored in one central place or data warehouse. It could be stored as unstructured data in your data lake or distributed over transactional and analytical systems [2]. Most of the time, spatial data is stored in geospatial information systems (GIS), which are separated from data warehouses. In this demo paper we describe how spatial and business data can be easily combined and analyzed over a landscape of heterogeneous systems. Combining this data and making it accessible to one SQL statement allows unified location-aware intelligence over all enterprise data.

This paper is organized as follows. In Section 2 we discuss the overview of Amazon Redshift [3] and its core spatial functionality. In Section 3 we describe the spatial support architecture. In Section 4 we describe the demo scenario and how we combine different data sources with federation and query pushdown to S3. Finally, we conclude in Section 5.

2 SYSTEM OVERVIEW

Amazon Redshift (AR) is a petabyte-scale cloud data warehouse. Data is stored in a columnar fashion and its architecture allows to use massive parallel processing (MPP). An AR cluster is composed of a *leader node* (LN), and one or more *compute nodes* (CNs). The LN receives the queries and creates their execution plans that often join data across various data sources such as Amazon S3 or Amazon Aurora PostgreSQL [5]. Once a distributed query execution plan is decided, the LN generates code for each query and distributes the executable to the CNs. The LN orchestrates the parallel execution of the plans, while the CNs execute the generated code of each query and return the results to the LN.

As of November 2019, AR is capable of storing, processing and querying two-dimensional spatial data, by introducing a new native data type called `GEOMETRY`. `GEOMETRY` is a first-class data type citizen, and the user is able to create columns that store `GEOMETRY` data, to query these columns, join them, apply spatial functions to them, *etc.* More than forty spatial functions are supported. These functions can take one or more geometry objects as an input, produce a new geometry object, or access attributes of geometric objects.

AR supports querying "live" data from Aurora and RDS PostgreSQL databases, including support for querying spatial

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06.
https://doi.org/10.1145/3318464.3384704

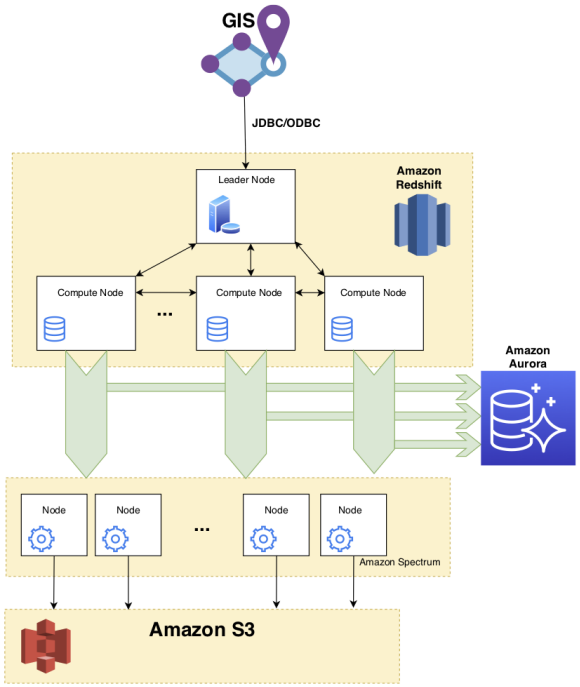

Figure 1: Amazon Redshift with Amazon Spectrum and Amazon Aurora

data stored in PostgreSQL. Since Redshift imports/exports geometries in Extended Well-Known Binary (EWKB) format, and spatial functions conform to OGC's Simple Feature Access, Redshift can also push spatial filters to PostgreSQL when applicable, effectively filtering the data as close to the source as possible, and reducing the amount of data transferred to Redshift for further processing.

This means that AR can be used as a single gateway to process and fetch spatial data from a transactional system like APG and from a data lake such S3 via Amazon Spectrum. GIS communicates with the LN only and transparently benefits from all three underlying engines with multiple cluster nodes, as shown in Figure 1. All access and processing of data stored in AR, Amazon Spectrum, and Amazon Aurora is federated and executed in parallel. All the above come to the GIS with no additional configuration or setup effort.

3 SPATIAL SUPPORT ARCHITECTURE

Amazon Redshift's Spatial support consists of three layers: the Geometry Engine (GE), the Spatial Integration (SI) and the Spatial Processing (SP) layer.

Geometry Engine layer. It deals with the fundamental operations on spatial objects. It is written in a generic fashion. It does not assume specific models/implementations for the geometric objects that it operates on, but is rather relying on concepts for these objects. These concepts refer to 2D, 3D, or 4D geometries. The algorithms are either applicable to concepts of specific dimension(s), or all possible available dimensions. The GE level currently supports 2D, 3D, and 4D geometries. In terms of geometry subtypes, we support *points*, *linestrings*, *polygons*, *multipoints*, *multilinestrings*, *multipolygons*, and *geometry collections*. For the spatial algorithms to work efficiently, it is assumed that we have constant-time access to the points of a geometry whose subtype is not a geometry collection. For geometry collections, the requirement is that a point inside the collection is accessible in time proportional to its nesting level within the collection.

Besides the specific subtypes for geometries, we also support the notion of an abstract geometry concept. All algorithms in the GE layer are designed to work with not only the specific subtype concepts, but also the abstract geometry concept. In fact, a geometry collection is seen as a vector of abstract geometry objects. Given that we want the GE layer to work seamlessly and generically over different possible implementations for the hierarchy of geometric types, we require from the user of the GE layer, via a traits mechanism, to provide the way to convert an abstract geometry object to one of the supported subtypes. This is used both to cast an input geometry, seen as an abstract geometry object, to its specific subtype, as well as to access the different geometry objects within a geometry collection. From the C++ point of view, we have designed this as a combination of compile-time tag dispatching and dynamic run-time casting. This is achieved by associating each geometric object (abstract or concrete) to a specific type via appropriately specialized meta-functions. When the type of the geometric object is known, tag dispatching is utilized to identify the proper implementation for the algorithm of interest. On the other hand, if the subtype of the object is not known at compile time (as in the case of abstract geometries, or for the objects within a collection), we discover its type at run-time and employ tag dispatching based on the subtype discovered.

Spatial Integration layer. It is specific to AR and it defines the models for the concepts to be used in the GE layer. Defining the models also involves defining the necessary meta-functions for these models to operate with the GE layer. These models are designed to meet the run-time requirements for accessing the points of the different geometry subtypes, as these are mandated by the GE layer. At the same time, they are designed to fit within the code generation framework of AR, and to be used as the memory representations of geometry objects throughout the system. This layer also provides all the functionality needed to implement the spatial functions supported in AR.

Spatial Processing layer. It is fully integrated within AR and is responsible for (a) defining the `GEOMETRY` data type

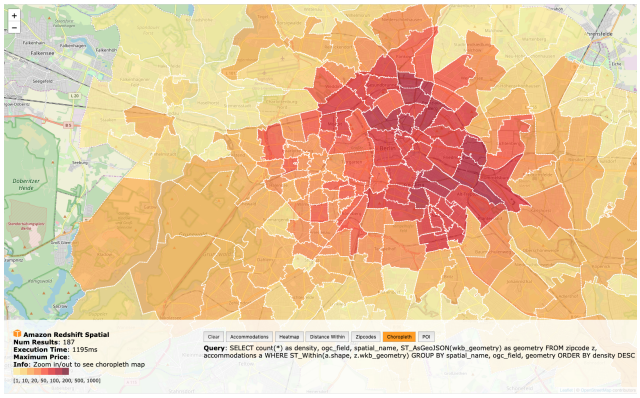

Figure 2: Demo cloud application.

in AR; (b) generating code in the context of spatial query execution; (c) supporting DDL and DML statements in tables with GEOMETRY columns; (d) materializing spatial columns within the system; (e) allowing for import and export of spatial columns via COPY/UNLOAD; (f) supporting system-wide operations, such as resizing clusters, *etc.* SP layer treats GEOMETRY columns as variable-size fields with a maximum size of approximately 1MB. We currently support only raw encoding for GEOMETRY columns, which is essentially the in-memory representation of the geometry objects.

4 LIVE DEMO SCENARIO

We use Amazon Elastic Compute Cloud (EC2) and Amazon Elastic Beanstalk for scalable computing capacity and fast development of applications. The application uses PHP and PDO [6] to connect against AR. Figure 2 shows a choropleth map, which colors the zip codes by cardinality of accommodations. The queries dynamically change depending on the use of the application, which allows us to interact with the attendees during the live demo and show different query execution and result visualization.

4.1 Datasets

The demo uses data stored in 3 tables. The first table contains 22K Airbnb accommodations in Berlin [1]. An accommodation has multiple attributes, such as location (2D point with longitude and latitude values), name, information about the neighbourhood, price, *etc.* The second table is an external table referencing an Aurora PostgreSQL database where we store 190 zip codes of Berlin each with a polygon, a name, and additional meta attributes. The polygon stores multiple 2D coordinates which define a closed ring. PostgreSQL comes with its own spatial engine (PostGIS [7]) and is optimized for OLTP, whereas Redshift is optimized for OLAP workloads. The third table is another external table, referencing a CSV-file stored on Amazon S3. The CSV-file contains 11M rows with point of interest (POI) data for the whole globe, publicly available on [4]. It includes double longitude and latitude coordinates along with additional attributes, such as the name and rating of the location.

4.2 Workload

The data is used to demonstrate three main aspects. First, the spatial core capabilities of AR. Spatial data is analyzed with SQL predicates and functions in combination with business data. This emphasizes the deep integration of spatial analytics into our distributed query execution engine. Listing 1 shows a spatial join where the zip code data is fetched out of Aurora PostgreSQL and the join condition is executed in AR.

Listing 1: Spatial Join

```
SELECT count(*) as density, spatial_name,
ST_AsGeoJSON(wkb_geometry) as geometry
FROM zipcode z, accommodations a
WHERE ST_Within(a.shape, z.wkb_geometry)
GROUP BY spatial_name, geometry
ORDER BY density DESC
```

Listing 2: Access data on S3 via an external Table

```
SELECT name, country, lat, long,
ST_SetSRID(ST_Point(long,lat),4326) as poi
FROM spectrum.geoname
WHERE ST_Within(poi,
ST_GeomFromText('POLYGON((...))', 4326))
```

Listing 3: Federated Query

```
SELECT ogc_field, spatial_name, zip_geometry
FROM aurora.zipcode
WHERE ST_XMax(wkb_geometry) > screen_xmin AND
ST_XMin(wkb_geometry) < screen_xmax AND
ST_YMax(wkb_geometry) > screen_ymin AND
ST_YMin(wkb_geometry) < screen_ymax
```

Second, the transparent integration with data stored in Amazon S3. Queries against the external table in Amazon S3 enable us to combine data from our object store with relational data stored in Redshift (see Listing 2).

Third, federated queries enable us to easily combine data and pushdown predicates to Aurora PostgreSQL. The pushdown happens when the predicate semantic is exactly the same. This allows us to integrate Aurora PostgreSQL as a new data source in the demo scenario without changing the result set. One example is described in Listing 3, with explain plan shown in Figure 3, where we use a fast bounding box check which is pushed down to Aurora.

The demo will show how an GIS analyst can combine all three data sources (Redshift, S3, Aurora PostgreSQL) to analyze the distribution of accommodations in relation to prices and zip codes. The live demo can be used to interact with the audience to get more insights about the data. Our goal is to simplify complex query execution over multiple heterogeneous data sources. Such a query that combines

```
XN PG Query Scan spatial_table  (cost=0.00..42.50 rows=1000 width=438)
->  Remote PG Seq Scan apg_beefy_gistest_public.spatial_table  (cost=0.00..30.00 rows=1000 width=438)
Filter: ((st_xmax(geom) > 13.1995582580566::double precision) AND
(st_xmin(geom) < 13.4603118896484::double precision) AND
(st_ymax(geom) > 52.4494187620065::double precision) AND
(st_ymin(geom) < 52.5305298636763::double precision))
```

Figure 3: Query plan for query in Listing 3.

Listing 4: Combined Query

```
SELECT a.name, a.price, ST_X(a.shape), ST_Y(a.shape), g.longitude, g.latitude
FROM public.accommodations a, zipcode z, spectrum.geoname g
WHERE ST_DistanceSphere(a.shape, ST_SetSRID(ST_Point(g.longitude,g.latitude), 4326)) < 500 AND
ST_Contains(z.wkb_geometry, a.shape) AND
ST_Contains(z.wkb_geometry, ST_SetSRID(ST_Point(g.longitude,g.latitude), 4326)) AND
ST_XMax(z.wkb_geometry) > 13.1995582580566 AND ST_XMin(z.wkb_geometry) < 13.4603118896484 AND
ST_YMax(z.wkb_geometry) > 52.4494187620065 AND ST_YMin(z.wkb_geometry) < 52.5305298636763;
```

```
XN Nested Loop DS_BCAST_INNER
  Join Filter: ((st_distancesphere("inner".shape, st_setsrid(st_point("outer".longitude, "outer".latitude), 4326))
               < 500::double precision) AND
            (st_contains("inner".wkb_geometry, st_setsrid(st_point("outer".longitude, "outer".latitude), 4326)) = true))
  ->  XN S3 Query Scan g
        ->  S3 Seq Scan spectrum.geoname g location:"s3://hinnerk/spatial/DE/allCountries.txt" format:TEXT
  ->  XN Materialize
        ->  XN Nested Loop DS_BCAST_INNER
              Join Filter: (st_contains("inner".wkb_geometry, "outer".shape) = true)
              ->  XN Seq Scan on accommodations a
              ->  XN Materialize
                    ->  XN PG Query Scan z
                          ->  Remote PG Seq Scan apggis.zipcode z
                                Filter: ((st_xmax(wkb_geometry) > 13.1995582580566::double precision) AND
                                         (st_xmin(wkb_geometry) < 13.4603118896484::double precision) AND
                                         (st_ymax(wkb_geometry) > 52.4494187620065::double precision) AND
                                         (st_ymin(wkb_geometry) < 52.5305298636763::double precision))
----- Nested Loop Join in the query plan - review the join predicates to avoid Cartesian products -----
```

Figure 4: Query plan for the combined query in Listing 4.

all three data sources can be seen in Listing 4, with explain plan shown in Figure 4. The plan consists of two nested-loop joins. The second join is between a Redshift table and an external spatial table in Aurora PostgreSQL (APG). Spatial filters have been pushed down to APG, while the join is on the spatial predicate ST_Contains. The first join is between the materialized result set of the second join and an external table in S3. The join is based on the conjunction of two spatial functions: the spatial predicate ST_Contains and a condition on the spherical distance between the geometries (points) of the Redshift table and the external S3 table.

5 CONCLUSION

In this demo paper we briefly describe Amazon Redshift's spatial support. We have considered spatial data sets coming from different sources (Redshift's native storage and external tables from S3 and APG), and have showed how to combine them to produce complex queries, including spatial joins. The queries are executed in the context of cloud applications and their results are visualized on top of a map.

REFERENCES

[1] Airbnb *Accommodation data for Berlin.* https://www.kaggle.com/brittabettendorf/berlin-airbnb-data#listings.csv

[2] V. Pandey, A. Kipf, T. Neumann, A. Kemper. *How Good Are Modern Spatial Analytics Systems?* Proceedings of the VLDB Endowment, Vol. 11, No. 11, 2018.

[3] *Amazon Redshift* https://docs.aws.amazon.com/redshift

[4] GeoNames *Point of interest database dump.* https://www.geonames.org/export/

[5] M. Cai, M. Grund, A. Gupta, F. Nagel, I. Pandis, Y. Papakonstantinou, and M. Petropoulos. *Integrated Querying of SQL database data and S3 data in Amazon Redshift.* IEEE Data Eng. Bull. 41(2), 2018.

[6] PDO *PHP Data Objects.* https://www.php.net/manual/en/book.pdo.php

[7] *PostGIS: Spatial and Geographic objects for PostgreSQL* https://postgis.net/