Synergy of Diversity: Data, Modeling and Decisions
Łódź University Press 2025, 47–75
https://doi.org/10.18778/8331-969-8-05

VARIOUS REPRESENTATIONS OF GEOMETRIC OBJECTS IN SPATIAL RELATIONAL AND SPATIAL OBJECT-RELATIONAL DATABASES MANAGEMENT SYSTEMS

ALEKSANDRA STASIAK

Abstract. In this paper we look at various kinds of representations of geometric objects in spatial relational (or object-relational) database management systems. We confront it with Simple Feature Access and SQL/MM (Part: 3) standards. We also try to find out some other representations of geometric objects that are used not only in traditional spatial database management systems but also in some NoSQL databases. We indicate functions and methods related to geometry representations such as WKT, WKB, GML, GeoJSON, GeoHash, KML, EWKT, EWKB, TWKB, HEXEWKB used in MySQL, Oracle XE and PostGIS/PostgreSQL. We illustrate this with examples, also showing some ideas for importing/exporting geometries between these database management systems.

1. Introduction

The establishing in 1999 by Open Geospatial Consortium of the standard for storing and accessing geospatial data in relational databases [30] and then the rise of appropriate ISO standards [21, 22] started the process aimed at achieving the interoperability of different spatial databases, that is, in particular, the ability to communicate and transfer data between these systems that requires the user to have little knowledge about their characteristic features, see [20]. The aforementioned standards (along with subsequent versions, see [41, 42]), force the database management systems that are compatible with these documents to be able to write to the databases spatial attributes of geospatial features given in a specific text format (called Well-known text, briefly: WKT) or in a special binary form (called Well-known binary, briefly: WKB) and also to convert spatial attributes stored already in the databases to these formats. Thanks to this, the transfer of spatial data between different databases has been significantly simplified. It also has started to be much easier to use an unfamiliar spatial relational database management system, because it is not necessary to know the methods of creating geospatial features specific for that system (so called native methods), it is enough to find the appropriate converting functions or constructors based on WKT or WKB representations. These representations are also described in the third part of SQL/MM standard, devoted to spatial data. This standard additionally gives one more possibility: a textual representation of geographic feature in GML language.

All these standards relate to relational databases – databases with roots dating back to the 1970s. They are also used by object-relational database management systems which where first created in the 1990s. But now there are many other models of data and various new database management systems, especially those from NoSQL and NewSQL families. Some of these systems also store and process spatial data, e.g MongoDB, CouchDB, Amazon DocumentDB (document stores), Cassandra (wide-column store), Neo4j (especially with Neo4j Spatial Library) (graph database), Redis (key-value store), Elasticsearch, Solr (search engines), Amazon DynamoDB with Geo Library (document store, key-value store), see [5, 13, 18, 65, 66]. There are papers in which authors attempt to compare geospatial capabilities of relational and NoSQL databases or analyse performance of the queries in such databases [7, 3, 4, 8, 59, 27] or even try to present some hybrid (NoSQL-SQL) approach [19]. Therefore a very important question arises whether it is possible to talk about the interoperability of traditional (i.e. spatial relational or spatial object-relational) and new types of databases. However, in this

2020 *Mathematics Subject Classification.* 68P20, *Computing Classification System*: H.3.3.
Key words and phrases. spatial databases, Simple Feature Access, SQL/MM Part 3: Spatial, representations of geometries.

paper we do not try to answer it (for instance authors of [6] showed some possibilities to interoperate spatial data stored in SQL and NoSQL databases via OGC services using GML format) or to consider these new types of systems or their capabilities, but we emphasise the fact that the representations of geometric objects used in them, especially GeoJSON, infiltrated the relational or object-relational databases and became an important way to represent such objects there. Extending the capabilities of relational or object-relational systems to the processing and use of these newer formats not only allows these systems to compete in some respects with NoSQL systems, but also could be an opportunity to build some interoperability and make the export and import of geospatial data easier. In Section 2 of the paper, we firstly look carefully at different versions of above-mentioned standards and try to find in them all methods and functions connected with representations of the spatial attributes of geospatial features. Then we point out various other representations of the geospatial data, such as GeoJSON or GeoHash. In Section 3, we will look at three of leading spatial relational / object-relational database management systems (Oracle, MySQL and PostgreSQL with PostGIS), checking if they implement the methods, converting functions and constructors required by the standards and whether they provide others representations, especially those that are also used in NoSQL systems. Section 4 includes some examples to demonstrate how the discussed functions and methods work.

2. Various representations of geometric objects

2.1. **Representations of geometric objects described in international standards connected with storing and retrieving spatial or geospatial data in relational databases.**

2.1.1. *"Simple Feature Access" Standard.* The first standard concerning spatial data in databases was OpenGIS Simple Features Specification For SQL, from 1999 (let us call it SF standard, for brevity) [30]. In 2005, it was superseded by OpenGIS Implementation Specification for Geographic information – Simple feature access, known widely as "Simple Feature Access" (briefly: SFA standard). The SFA standard consists of two parts – Part 1: Common architecture [41] and Part 2: SQL option [42] and is compatible with ISO 19125 standard [21, 22]. In these standards the geospatial features have both spatial and non-spatial attributes. The values of the spatial attributes are described in terms of geometries – geometric objects such as points, curves, and surfaces (called geometric primitives) or homogeneous or heterogeneous collections of such elements. The so-called simple features are based on two-dimensional geometries with linear interpolation between vertices. Each geometric object is associated with a number identifier, called Spatial Reference System ID (abbreviated as SRID), which identifies the coordinate space in which the object is defined. The documents are based on an extended Geometry Model, in which geometries form an appropriate hierarchy of classes, with Geometry class as a root. Some of these classes are non-instantiable ones – it is impossible to create their instances, they only serve as base classes for their subclasses and define a set of methods for them to inherit. Standards describe the classes in detail along with the attributes, methods and assertions for each class. They also provide the Dimensionally Extended Nine-Intersection Model, some methods for testing spatial relations between geometric objects, some methods that support spatial analysis, WKT and WKB representations of geometric objects, Well-known Text Representation of Spatial Reference Systems and define sets of supported units, spheroids, geodetic datums, prime meridians, and map projections.

The tables whose rows contain geospatial features are called feature tables. Each feature is stored in one row of the table. The non-spatial attributes of features are stored in columns of a feature table whose types are drawn from the set of standard SQL data types. The way the spatial attributes are stored depends on a chosen implementation. The standards propose three possible ways (all with ODBC access in SF standard or SQL/CLI access in SFA). In the first two cases, geometries are stored in an additional table (called a geometry table) by means of numeric (respectively: binary) SQL types (using WKB representation of geometry in the second case). In the third case, geometries are stored in the feature table using SQL Geometry types – types that extend the standard set of SQL data types via Abstract Data Types and are based on the extended Geometry Model. In all these cases, standards require the existence of appropriate metadata views: one containing information about available coordinate systems and the second containing information about feature tables stored in the database. Both SF and SFA standards tell us about two representations of geometry that can be used both to create new instances and to convert existing ones: the text representation (so-called Well-known Text representation, briefly: WKT representation) and binary representation (so-called Well-known Binary representation, briefly: WKB representation). Thanks to the WKB representation, the geometry is presented as a contiguous stream of bytes. This stream arises by changing to bytes a sequence of numbers describing a given geometry and the way it is being described. In particular numbers indicate the order of bytes in the stream, the geometry type and the geometry coordinates. The Well-known Text representation of geometry

presents it as a text composed of type names, brackets, commas and numbers indicating geometry coordinates. The way the text should be created is described by special grammar rules described in BNF notation. Both of these representations do not include information about the coordination system and both are defined for the geometric elements described in a given standard, formally for each geometry type, but in practice the grammar or structure of it is defined only for geometries belonging to instantiable subclasses of the Geometry class.

Therefore the SF standard describes details of these representations for zero-, one- or two-dimensional geometries existing in two-dimensional coordinate space, belonging to Point, LineString, Polygon, GeometryCollection, MultiPoint, MultiLineString or MultiPolygon class, assuming linear interpolation between vertices of the elements. Among the methods for the Geometry class there are methods exporting a given geometry to WKT or WKB representation (`AsText` and `AsBinary` method, respectively).

In order to be compliant with the SF standard a database management system should implement functions having capabilities such as those described in Table 1. Systems can also implement some optional functions that support Polygon or MultiPolygon type elements creation given an arbitrary collection of possibly intersecting rings or closed LineString values (see Table 2).

In the first version of SFA standard (version 1.1 from 2005, [31, 32]), the coordinate dimensions of geometries, the hierarchy of classes and the functions connected with WKT and WKB formats remained the same as in SF. In version 1.2.0 (from 2006, [33, 34]) and in the latest one (version 1.2.1 from 2010–2011, [36, 35]), the instantiable subclasses of Geometry class are restricted to zero-, one- or two-dimensional geometric objects that exist in two-, three- or four-dimensional coordinate space (one can also consider the altitude and/ or the measure and therefore achieve coordinate dimension of geometries equal to 3 or 4). There is also a wider set of subclasses with Triangle, Polyhedral Surfaces and TIN; all still limited to the linear interpolation between vertices in all existing subtypes. The Polyhedral Surface class is mentioned in both parts of the standards, the Triangle and TIN classes - only in the first parts. Part 1 also defines semantics for WKT and WKB representations of geometric objects for all instantiable classes and presents `AsText` and `AsBinary` methods. Part 2, related to SQL-implementation, presents the SQL routines:

- `ST_AsText` and `ST_AsBinary` (for obtaining, respectively, the WKT and WKB representation of a geometric object),
- `ST_WKTToSQL` and `ST_WKBToSQL` (used to construct geometric objects from, respectively, their WKT and WKB representation).

These routines are specified in SQL/MM.

2.1.2. *SQL/MM Standard.* SQL/MM (actually: SQL Multimedia and Application Packages) is an ISO standard (ISO/IEC 13249-3) with Part 3: Spatial, which defines the way of storing, retrieving and processing spatial data using the SQL language. The first version of that standard was published in 1999. It is based on a hierarchy of geometry types, similar to the hierarchy of geometry classes defined in SFA standard, but with more types. In particular in the latest, fifth edition from 2016 [24], there are also types for different kinds of curves (for instance: geodesic curves and NURBS), types for geometries with various kinds of interpolation between vertices (for instance: elliptical, clothoid and spiral), a type for circles, types for geometries with vertices connected by circular arcs or by both circular arcs and line segments, types for some kinds of solids and compound surfaces (that is instantiable types such as: ST_GeodesicString, ST_NURBSCurve, ST_EllipticalCurve, ST_Clothoid, ST_SpiralCurve, ST_Circle, ST_CircularString, ST_CompoundCurve, ST_CurvePolygon, ST_BRepSolid, ST_CompoundSurface). The instantiable subtypes of the ST_Geometry type (the root class of the hierarchy) are, as in later version of SFA standard, zero-, one- or two-dimensional geometric objects that exist in two-, three- or four-dimensional coordinate space.

The SQL/MM standard describes:

- `ST_WKTToSQL`, `ST_WKBToSQL`, and `ST_GMLToSQL` methods returning a geometry value for a given WKT / WKB / GML representation, respectively, and `ST_AsText`, `ST_AsBinary`, `ST_AsGML` methods that provide respectively the WKT /WKB / GML representation of a given geometry,
- functions `ST_GeomFromText`, `ST_GeomFromWKB`, `ST_GeomFromGML` taking a WKT / WKB / GML representation of a geometry or such representation and a SRID identifier and returning a geometry value for that representation,

- three functions for each instantiable type that take respectively a WKT / WKB / GML representation of a geometry of an appropriate type (or a representation and a SRID identifier) and return a geometry value of an appropriate type (e.g. `ST_PointFromText`, `ST_PointFromWKB`, `ST_PointFromGML`, `ST_LineFromText`, `ST_LineFromWKB`, `ST_LineFromGML`, `ST_CircularFromTxt`, `ST_CircularFromWKB`, `ST_CircularFromGML`, etc.) and additionally functions `ST_BdPolyFromText`, `ST_BdPolyFromWKB`, and `ST_BdMPolyFromText`, `ST_BdMPolyFromWKB` that take a WKT / WKB representation of geometry of ST_MultiLineString value (or such representation and a SRID) and return a geometry value of ST_Polygon (respectively: ST_MultiPolygon) type.

That means that SQL/MM standard uses not two, as SF/SFA standards, but three kinds of representation of geometries: WKT, WKB and GML. The GML is an XML-based language (i.e. a markup language) designed for expressing geographical features (see [39, 29, 10]). It provides a textual representation that can be used for storing or transporting geometries. GML is defined by Open Geospatial Consortium and OGC maintains its standards. In 2007 GML was also adopted as an International Standard (ISO 19136:2007) [23]. The latest ISO standard for GML was established in 2020 [25].

It is also worth mentioning that not only RDBMSs use discussed formats – or instance there are some NoSQL databases that also support WKT format (e.g. Elasticsearch, Solr or Neo4j) or even store geospatial features in WKT (e.g. DataStax Enterprise 6.0 based on Cassandra [11]).

Let us also notice that names of all types, methods, and functions (as well as tables and views) connected with spatial data defined in SQL/MM have the prefix ST, originally stood for "Spatial and Temporal", now interpreted as "Spatial Type". The same prefixes can be found in the newest versions of SFA standard, which shows that these documents refer to the SQL/MM document. A comparison of Simple Feature Access/SQL and SQL/MM – Spatial (2003) can also be found in part 2 of SFA (see Table B 1 in [35]). The SQL/MM standard also describes some other features such as Topology-Geometry and Topology-Network models and types, functions and methods connected with Linear Referencing System. Since not only geometries or spatial references systems have a well-known representation (but also elements of types related to angles, directions, vectors and linear referencing system), therefore the document contains specification of some other functions constructing appropriate elements from its WKT, WKB or GML representation.

At the end of this section, let us note a problem that arises despite the existence of the mentioned standards - namely the problem with the order (or else with an interpretation) of the X and Y values in coordinate tuples, especially in the case of geographic coordinate systems. In SF/SFA standards there is no information about the meaning of the X and Y coordinate in WKT representation, so there is no general guideline as to whether latitude or longitude goes first. That problem is wider, it concerns not only the WKT representation of geometries, but many domains recording the position of objects (see [50, 14]) and can cause troubles especially during exporting and importing data between different systems. The standards give only one clue: it is the SRID identifier that gives meaning to the coordinates and allows to interpret these values properly (see 6.2.3 in [35]).

2.2. **Other representations of geospatial features.** The WKT, WKB and GML formats are not the only ways to represent the geospatial features. The NoSQL databases most often (or mainly) use a different representation – GeoJSON format. GeoJSON is an open standard format that is intended for encoding and interchanging the geographic features [1]. It's latest standard specification (The GeoJSON Specification (RFC 7946)) comes from 2016 [9]. GeoJSON is based on JSON (JavaScript Object Notation) – lightweight, text-based, language-independent data-interchange format derived from JavaScript [2, 12]. JSON is widely used in many NoSQL databases, there are systems based on storing data just in this format (e.g. MongoDB), but for a few years it is also adopted by relational and object-relational databases, that can now store, index and query JSON data (for instance MySQL, Oracle, PostgreSQL, SQL Server, etc.) using native or new, dedicated, data types. In certain applications, JSON has replaced XML, in documentation and on websites (see [43, 26, 58] for instance) some comparison between JSON and XML can be found, with pros and cons of both these solutions. GeoJSON supports geometries with some additional properties (so-called Feature objects), the sets of features (the FeatureCollection objects) and Point, LineString, Polygon, MultiPoint, MultiLineString, and MultiPolygon types. There is also an extension of GeoJSON format, called TopoJSON [15], that is designed for geospatial topologies encoding. It is worth mentioning that GeoJSON is not maintained by any standards organization.

Another format used both in relational, object-relational and NoSQL databases is GeoHash. GeoHash system [64] encodes a geographic location given by latitude/longitude as a string of letters and digits. It uses space-filling curves, transforms the latitude and longitude coordinates space into a hierarchical discrete grid,

and assigns the location values 0 or 1 depending on which grid fields it belongs to on the subsequent levels. Then the string of bits is transformed into a shorter string composed of alphanumeric characters. The resulting string represents a rectangular area, the precision of rectangle increases with the length of the string. This technique also allows to move from two-dimensional to one-dimensional data and is used to build so called geohash-based spatial indexes (see [61, 28], for instance).

Some database systems (e.g. Oracle, PostGIS/PostgreSQL or Splunk) also provide support for a KML format. KML (Keyhole Markup Language) is a XML-based language used to display geographic data in Earth browsers, originally developed for Google Earth, but now used in many other tools and mapping websites [40]. It has been an international standard of the OGC since 2008.

3. Functions and methods connected with various representations of geometric objects in MySQL, Oracle and PostGIS/PostgreSQL

The first four places in the popularity ranking of relational databases prepared on the db-engines.com website (see [60]) have been occupied for many years by Oracle, MySQL, Microsoft SQL Server and PostgreSQL. Each of these systems is able to store and process spatial data or has special modules, extensions or tools which are able to do that. Let us check what functions and methods intended for representing geometric objects can be found in selected three of these systems (together with modules, extensions, and tools). Let us consider MySQL RDBMS (version 8), Oracle XE with Oracle Spatial and Graph (version 21c) and PostgreSQL (version 14) with PostGIS module (version 3.4).

3.1. **MySQL.** The open-source relational database management system MySQL follows SFA standard (Part 2: SQL Option) (previously followed SF), but it is not fully compatible with any of its versions (see Section 11.4 in [47]). It implements only a subset of a set of geometry types described in SFA, with geometric objects that exist only in two-dimensional coordinate space, and it does not implement the whole required functionality. It enables the creation, storage and analysis of geographic features in four of its engines: MyISAM, InnoDB, ARCHIVE and NDB (with spatial indexes in the first two of these engines). For many years MySQL did not support different coordinate systems and did not provide appropriate metadata views. Even if it was possible to assign to the geometry a SRID identifier, it did not affect anything, the calculations were done assuming Cartesian coordinates. In version 8.0 the things have changed: metadata views (for coordinate systems and feature tables) have been added, as well as a support for geographic computations [56, 57]. Tables, spatial indexes and functions are now "SRID aware" (the InnoDB engine permits both Cartesian and geographic coordinates systems now, MyISAM only Cartesian ones). Most spatial functions (but not all) are able to do proper calculations in case of geographic coordinates [56]. The database documentation [47] says that both SFA and SQL/MM standards are important for MySQL implementation of spatial operations, and for implementation of functions and structure of the metadata views.

MySQL has spatial data types that correspond to SF classes, that is: GEOMETRY, POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON and GEOMETRYCOLLECTION. MySQL supports standard spatial data formats: WKT and WKB (see Table 3 for functions associated with these two formats). Internally, it stores geometry similarly to WKB format, but with an initial 4 bytes to indicate the SRID value. MySQL also ensures functions connected with GeoJSON and GeoHash representations (see Table 4). However, it lacks functions connected with GML and KML.

3.2. **Oracle object-relational database.** Oracle enables spatial data storing and processing thanks to Oracle Spatial and Graph [48, 49, 44, 45, 46] – a module that includes advanced features not only for vector spatial data with spaghetti data model, but also for topologies, networks, rasters and social and semantic graphs. The module contains MDSYS schema that defines the storage, syntax, and semantics of supported geometric data types, spatial indexing mechanism, Linear Referencing System, support for topological and network models, GeoRaster, support for TIN and NURBS and subprograms for performing spatial queries, spatial analysis and some tuning and utility works. Until December 2019, the full set of Oracle Spatial and Graph capabilities was available only as a paid option in Oracle Database Enterprise Edition, whilst a selected part of it – collected in the so-called Oracle Locator – was available in other versions, especially in a free Oracle Database Express Edition. Since that time all Oracle Spatial and Graph capabilities are available with all editions of Oracle Database, and information about Oracle Locator has been removed from the documentation (see Section Changes in Oracle Database Release 21c in [49]). Oldest versions of Oracle Spatial and Graph (starting with Oracle Database release 10g (version 10.1.0.4) implemented SF standard (with Geometry Types Implementation) (see Section 1.21 in [49]). The newer versions (18c, 19c) are compliant with Part 2 of the SFA

Types and Functions, version 1.1, Oracle Spatial 21c is compliant with Types and Functions version 1.2.1 [38]. Oracle Spatial and Graph also supports the use of types specified in SQL/MM, Part 3. (see the beginning of Chapter 3 in [49]).

Oracle Spatial and Graph provides two geometry types: SDO_GEOMETRY (its native type) and ST_GEOMETRY (based internally on SDO_GEOMETRY, but with many additional static and member functions). The documentation (see Section 3.1 in [49]) says that the Oracle Spatial and Graph SDO_GEOMETRY and the ST_GEOMETRY type from SQL/MM standard with all its subtypes (ST_POINT, ST_CURVE, ST_LINESTRING, ST_CIRCULARSTRING, ST_COMPOUNDCURVE, ST_SURFACE, ST_POLYGON, ST_CURVEPOLYGON, ST_GEOMCOLLECTION, ST_MULTIPOINT, ST_MULTICURVE, ST_MULTILINESTRING, ST_MULTISURFACE, ST_MULTIPOLYGON) are essentially interoperable. The list of SQL/MM functions compared with their counterparts from Oracle Spatial and Graph can be found in Section 3.2 in [49].

The easiest way to obtain a WKT, WKB, GML, GeoJSON or KML representation of a geometry stored as SDO_GEOMETRY is to use object's methods such as:

- `GET_WKT` (returning a CLOB),
- `GET_WKB` (returning a BLOB),
- `GET_GML`, `GET_GML311` and `GET_GML321` (returning a CLOB, all with no input arguments or with SRSNAMESPACE and SRSNSALIAS input arguments (as VARCHAR2) or with COORDORDER input argument (as NUMBER) or with SRSNAMESPACE, SRSNSALIAS and COORDORDER input arguments),
- `GET_GEOJSON` (returning a CLOB),
- `GET_KML` (returning a CLOB).

To create an SDO_GEOMETRY object from well-known textual or binary representations of a geometry, one can use:

- a constructor for that type taking a WKT (as a CLOB) and a SRID (as a NUMBER, with default NULL value),
- a constructor for that type taking a WKT (as a VARCHAR2) and a SRID (as a NUMBER, with default NULL value),
- a constructor for that type taking a WKB (as a BLOB) and a SRID (as a NUMBER, with default NULL value).

But one can also take advantage of functions related to the WKT, WKB, GML, GeoJSON, JSON or KML representation of geometries included in SDO_UTIL package (see Table 5 and Table 6).

Oracle Spatial and Graph also contains functions that validate geometries inputs in WKT and WKB formats: `SDO_UTIL.VALIDATE_WKTGEOMETRY` and `SDO_UTIL.VALIDATE_WKBGEOMETRY` (the first taking a WKT representation as a CLOB or as a VARCHAR2 and the second taking a WKB representation of geometry as a BLOB). These functions return string TRUE if the input is valid (i.e. have the appropriate syntax, consistent with OGC documents) and FALSE otherwise.

Oracle also supports geohashes through subprograms from SDO_CS package (see Table 7). Unfortunately tests done by the author in Oracle XE 21c database (using SQL* Plus) show that the `SDO_CS.FROM_GEOHASH` function returns geometry with invalid GTYPE and needs some corrections. Package SDO_CS also includes `SDO_CS.GET_GEOHASH_CELL_HEIGHT` and `SDO_CS.GET_GEOHASH_CELL_WIDTH` subprograms that take a length of the geohash string and return the cell height (the cell width, respectively) of that geohash.

As mentioned at the beginning of this section, Oracle Spatial also has a second type for storing geometries, namely ST_GEOMETRY type. This type also has methods connected with the representations of geometries, such as:

- `GET_WKB` (returning a BLOB),
- `GET_WKT` (returning a CLOB),

and static functions such as:

- `FROM_WKT` taking a WKT representation (as a CLOB or a VARCHAR2) or WKT representation and a SRID (as a NUMBER) and returning an ST_GEOMETRY object,
- `FROM_WKB` taking a WKB representation (as a BLOB) or a WKB representation and a SRID (as a NUMBER) and returning an ST_GEOMETRY object,

- `ST_ASTEXT` taking a geometry (as a ST_GEOMETRY) and returning a WKT representation (as a CLOB),
- `ST_ASBINARY` taking a geometry (as a ST_GEOMETRY) and returning a WKB representation (as a BLOB),
- `ST_GEOMFROMTEXT` taking a WKT representation (as a CLOB) or a WKT representation and a SRID (as a NUMBER) and returning an ST_GEOMETRY object,
- `ST_GEOMFROMWKB` taking a WKB representation (as a BLOB) or a WKB representation and a SRID (as a NUMBER) and returning an ST_GEOMETRY object.

Moreover for the elements of ST_GEOMETRY type one can also use functions:

- `OGC_ASBINARY` (returning a BLOB with a WKB representation) and `OGC_ASTEXT` (returning a VARCHAR2 with a WKT representation),
- `OGC_POINTFROMTEXT`, `OGC_LINESTRINGFROMTEXT`, `OGC_POLYGONFROMTEXT`, `OGC_MULTILINESTRINGFROMTEXT`, `OGC_MULTIPOLYGONFROMTEXT` (taking a WKT representation (as a VARCHAR2) and a SRID (as a NUMBER(38), default NULL) and returning an ST_GEOMETRY object),
- `OGC_POINTFROMWKB`, `OGC_LINESTRINGFROMWKB`, `OGC_POLYGONFROMWKB`, `OGC_MULTILINESTRINGFROMWKB`, and `OGC_MULTIPOLYGONFROMWKB` (taking a WKB representation (as a VARCHAR2) and a SRID (as a NUMBER(38), default NULL) and returning an ST_GEOMETRY object),

belonging to MDSYS schema.

Summarizing, it follows that Oracle Spatial supports WKT, WKB, GML, KML, JSON, GeoJSON and GeoHash formats (with some problems with the last one). It is worth mentioning that when displaying the contents of a geometric column, Oracle uses either a text containing a call to the native constructor of SDO_GEOMERTY type (for instance in SQL Developer or SQL* Plus) or a GeoJSON format (for instance in Oracle Live SQL). That native constructor call (containing the name of the type, brackets, commas and values for object's fields: SDO_GTYPE, SDO_SRID, SDO_POINT, SDO_ELEM_INFO, SDO_ORDINATES) in author's opinion can be treated as the next representation of geometry, specific to this system only.

3.3. **PostgreSQL & PostGIS.** PostgreSQL, an open-source object-relational database management system, possesses its own, very specific set of a geometry types, with point, line, lseg, box, (open and closed) path, polygon, and circle type [62]. The elements of these types exist in two-dimensional coordinate space. Each type has a few textual representations for its elements - they also could be recognized as a type of geometry representation (very specific one, with many possible ways to describe one given element). PostgreSQL provides geometric functions and operators for these elements. However, it does not provide any metadata and coordinate systems, and does not follow any standards. Nevertheless, the whole thing changes when we start to use PostgreSQL together with PostGIS. PostGIS is an extension to the PostgreSQL DBMS which enables it to store, index, manage and analyse geographic data due to the standards and provides a lot of useful functions and procedures [53, 55]. As of version 0.9, released in September 2004, it supports all objects and functions included in SF standard (see Section 4.1 in [52]). PostGIS 3.0 (Crunchy Data) is compliant with SFA - Part 2 (versions 1.1 and 1.2.1) Types and Functions [37] (same was PostGIS 2.5). Moreover, PostGIS definitely extends that standard because it supports TIN and Polyhedral Surfaces (from version 2.0), allows geometries compound of arcs or containing both arc and linear segments (CIRCULARSTRING, COMPOUNDCURVE, CURVEPOLYGON, MULTICURVE, MULTISURFACE), provides some functionality connected with Linear Referencing System, with topologies and rasters. PostGIS also follows SQL/MM standard. Each of the functions or methods implementing elements from standards are provided with an appropriate precise comment in the documentation.

PostGIS possesses two basic types for geometric object – geometry (older one, with more functions defined on it) and geography. Calculations on elements written using geometry type are done in Cartesian coordinate system. Geography type uses geographic coordinates and supports only geometries from POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON and GEOMETRYCOLLECTION classes. It does not support curves, and elements of TIN or POLYHEDRALSURFACE class. PostGIS also has types for boxes: box2d and box3d, and geometry_dump type – a composite type used to describe the parts of complex geometry. As compatible with the SFA/SF standards, PostGIS not only supports WKT and WKB formats, but also extends them to EWKT and EWKB formats (Extended Well-Known Text/Binary) that contain SRID information (which standards do not allow). EWKB and EWKT formats are used for the "canonical forms" of PostGIS data objects. The documentation (see Section 4.2.1 in [54]) says that every valid OGC WKB/WKT is also valid EWKB/EWKT, however one should not rely on this, because it can change

in the future. PostGIS also uses HEXEWKB representation, that is in the hexadecimal numeral system (i.e. hex-encoded EWKB) and TWKB representation (Tiny Well-Known Binary, that is a compressed binary format aimed at minimizing the size of the output).

PostGIS provides many functions connected with WKT, EWKT, WKB, EWKB, HEXEWKB and TWKB formats (see Table 8 and Table 9). Let us notice that some of the presented functions (for instance `ST_GeomFromEWKT`) support types such as Circular Strings and Curves, Polyhedral Surfaces, Triangles and TINs. PostGIS also manages functions connected with GML (with support for versions 2 and 3), with GeoJSON, KML and GeoHash, see Table 10.

PostGIS also has some others geometry input or output functions that one can also consider as certain kinds of representations, for instance:

- `ST_AsLatLonText` taking a point in a latitude/longitude projection (as a geometry value) and a parameter describing the output format and returning a text with the Degrees, Minutes, Seconds representation of the given point due to the required format,
- `ST_AsEncodedPolyline` taking a Linestring geometry value (as a geometry type) and some optional parameter and returning text containing the geometry as an Encoded Polyline (string with series of coordinates produced by certain lossy compression algorithm [17]),
- `ST_LineFromEncodedPolyline` taking a text with Encoded Polyline and creating a LineString from it (as a geometry type),
- `ST_AsX3D` taking a geometry value (as a geometry type) and two optional parameters and returning a text with a geometry in X3D xml node element format,
- `ST_AsSVG` taking a geometry value (as a geometry type) and two other optional integer parameters and returning the geometry as Scalar Vector Graphics (SVG) path data (as a text).

One can see that PostGIS with PostgreSQL provides the greatest number of supported formats including WKT, WKB, EWKT, EWKB, TWKB, HEXEWKB, GML, GeoJSON, GeoHash and KML.

A summary for all systems discussed can be found in Table 11.

4. Examples

Let us consider a table CITIES storing information about 10 cities of Poland with the largest population (in 2021). The columns of the table are:

- id (primary key of the table, storing integers),
- name (column with not null constraint, storing variable-length character strings with maximum 50 characters),
- population (column with not null constraint, storing integers),
- location (column with not null constraints, storing geometries).

We regard each city as a point geometry, built on the basis of the given latitude and longitude values.

Firstly let us use the EXAMPLE database created in MySQL DBMS (version 8.0.27, installed within WampServer 3.2.6) by executing in MySQL console a statement shown in Listing 1.

Listing 1. Creating the EXAMPLE database in MYSQL

```
CREATE DATABASE examples;
```

In that database let us create the table cities (with AUTO_INCREMENT attribute on id column), using InnoDB engine and WGS84 coordinate system having SRID equal to 4326, see Listing 2.

Listing 2. Creating the cities table

```
USE examples;

CREATE TABLE cities (
id INTEGER AUTO_INCREMENT,
name VARCHAR(50) NOT NULL,
population INTEGER NOT NULL,
location GEOMETRY NOT NULL SRID 4326,
CONSTRAINT cities_PK PRIMARY KEY(id)
) engine=INNODB;
```

Choosing the definition of the coordinate system with an appropriate SRS_ID (see Listing 3), we can find out that the axis with a latitude value precedes in that coordinate system the axis with a longitude value.

Listing 3. Definition of the coordinate system with SRS_ID

```
SELECT definition
FROM INFORMATION_SCHEMA.ST_SPATIAL_REFERENCE_SYSTEMS
WHERE SRS_ID = 4326;

+-----------------------------------------------------------------------+
|DEFINITION                                                             |
+-----------------------------------------------------------------------+
| GEOGCS["WGS 84",DATUM["World Geodetic System 1984",SPHEROID["WGS 84",6378137, 298.257223563,
    AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]],PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901
    "]],UNIT["degree",0.0174532925199433278, AUTHORITY["EPSG","9122"]],AXIS["Lat",NORTH],AXIS["Lon",
    EAST],AUTHORITY["EPSG","4326"]]                                    |
+-----------------------------------------------------------------------+
1 row in set (0.05 sec)
```

Let us now fill the table cities with the data obtained on the basis of the information contained in [16] (population of the cities) and [51] (latitude and longitude of the cities). Using given latitude and longitude of the cities let us build the appropriate WKT representations and take advantage of `ST_PointFromText` function that converts a WKT representation to POINT geometry, see Listing 4.

Listing 4. Insert statements to the table cities in MySQL database

```
INSERT INTO cities (name , population , location) VALUES
('Warszawa', 1860281, ST_PointFromText('POINT(52.12 21.02)', 4326)),
('Kraków', 800653, ST_PointFromText('POINT(50.03 19.57)', 4326)),
('Wrocław', 672929, ST_PointFromText('POINT(51.07 17.02)', 4326)),
('Łódź', 670642, ST_PointFromText('POINT(51.47 19.28)', 4326)),
('Poznań', 546859, ST_PointFromText('POINT(52.25 16.55)', 4326)),
('Gdańsk', 486022, ST_PointFromText('POINT(54.22 18.38)', 4326)),
('Szczecin', 396168, ST_PointFromText('POINT(53.26 14.34)', 4326)),
('Bydgoszcz', 337666, ST_PointFromText('POINT(53.07 18.00)', 4326)),
('Lublin', 334681, ST_PointFromText('POINT(51.14 22.34)', 4326)),
('Białystok', 294242, ST_PointFromText('POINT(53.08 23.10)', 4326));
COMMIT;
```

After executing the INSERT statement, we can verify that all geometries are valid ones, see Listing 5.

Listing 5. Validation of the geometries from the table cities

```
SELECT c.id, ST_IsValid(c.location)
FROM cities c
ORDER BY c.id ASC;

+----+-------------------------+
| id | ST_IsValid(c.location) |
+----+-------------------------+
|  1 |                       1 |
|  2 |                       1 |
|  3 |                       1 |
|  4 |                       1 |
|  5 |                       1 |
|  6 |                       1 |
|  7 |                       1 |
|  8 |                       1 |
|  9 |                       1 |
| 10 |                       1 |
+----+-------------------------+
10 rows in set (0.00 sec)
```

We can also check the metadata connected with this geometry layer, see Listing 6. This metadata was completed automatically by the system.

Listing 6. Checking the metadata for the table cities

```
SELECT *
FROM INFORMATION_SCHEMA.ST_GEOMETRY_COLUMNS
WHERE TABLE_NAME='cities';

+----------+----------+--------+----------+-------+------+--------------+
|TABLE_CATALOG |TABLE_SCHEMA |TABLE_NAME |COLUMN_NAME  |SRS_NAME  |SRS_ID |GEOMETRY_TYPE_NAME |
+----------+----------+--------+----------+-------+------+--------------+
| def      | examples | cities | location | WGS 84| 4326 | geometry     |
+----------+----------+--------+----------+-------+------+--------------+
1 row in set (0.00 sec)
```

Now let us display the names of the cities, their population, appropriate WKT representations of their locations (using `ST_AsText` conversion function) and latitude and longitude values related to these geometries (let us notice that in WKT representation in MySQL the latitude is given first by default). The results of the query are ordered in descending order by population size, see Listing 7.

Listing 7. Displaying the WKT representation of geometries from the table cities along with latitude and longitude values related to their location

```
SELECT c.name NAME, c.population POPULATION, ST_AsText(c.location) WKT,
      ST_Latitude(c.location) LATITUDE, ST_Longitude(c.location) LONGITUDE
FROM cities c
ORDER BY c.population DESC;

+-----------+------------+--------------------+----------+-----------+
| NAME      | POPULATION | WKT                | LATITUDE | LONGITUDE |
+-----------+------------+--------------------+----------+-----------+
| Warszawa  |    1860281 | POINT(52.12 21.02) |    52.12 |     21.02 |
| Kraków    |     800653 | POINT(50.03 19.57) |    50.03 |     19.57 |
| Wrocław   |     672929 | POINT(51.07 17.02) |    51.07 |     17.02 |
| Łódź      |     670642 | POINT(51.47 19.28) |    51.47 |     19.28 |
| Poznań    |     546859 | POINT(52.25 16.55) |    52.25 |     16.55 |
| Gdańsk    |     486022 | POINT(54.22 18.38) |    54.22 |     18.38 |
| Szczecin  |     396168 | POINT(53.26 14.34) |    53.26 |     14.34 |
| Bydgoszcz |     337666 | POINT(53.07 18)    |    53.07 |        18 |
| Lublin    |     334681 | POINT(51.14 22.34) |    51.14 |     22.34 |
| Białystok |     294242 | POINT(53.08 23.1)  |    53.08 |      23.1 |
+-----------+------------+--------------------+----------+-----------+
10 rows in set (0.00 sec)
```

In Listing 8, let us display the names of the three cities with the largest population together with the GeoJSON representation of their locations.

Listing 8. Displaying the GeoJSON representation of the locations of the three cities with the largest population

```
SELECT c.name NAME, ST_AsGeoJSON(c.location) GEOJSON
FROM cities c
ORDER BY c.population DESC
LIMIT 3;

+----------+-------------------------------------------------+
| NAME     | GEOJSON                                         |
+----------+-------------------------------------------------+
| Warszawa | {"type": "Point", "coordinates": [21.02, 52.12]} ||
  Kraków   | {"type": "Point", "coordinates": [19.57, 50.03]} ||
  Wrocław  | {"type": "Point", "coordinates": [17.02    51.07]} |
+----------+-------------------------------------------------+
3 rows in set (0.00 sec)
```

By using `ST_AsBinary` function, let us find the WKB representation of the city of Warszawa (Warsaw – the capital of Poland) (see Listing 9), and using `ST_GeoHash` function (along with `ST_Longitude` and `ST_Latitude` functions) – the names and a geohash values for the cities with the population greater then 1 000 000 (see

Listing 10). Checking this geohash value in any website converter, for instance in [63], we actually will result in a location in the capital of Poland.

Listing 9. Displaying the WKB representation of the city of Warszawa

```
SELECT ST_AsBinary(c.location) WKB
FROM cities c
WHERE c.name='Warszawa';

+--------------------------------------------+
| WKB                                        |
+--------------------------------------------+
| 0x01010000008FC2F5285C0F4A4085EB51B81E053540 |
+--------------------------------------------+
1 row in set (0.00 sec)
```

Listing 10. Displaying the geohash value for the locations of cities with population over 1 000 000

```
SELECT c.name NAME,
ST_GeoHash(ST_Longitude(c.location), ST_Latitude(c.location), 20) GeoHash
FROM cities c
WHERE c.population>1000000;

+----------+----------------------+
| NAME     | GeoHash              |
+----------+----------------------+
| Warszawa | u3qbw24m180n27ys6p1h |
+----------+----------------------+
1 row in set (0.00 sec)
```

Now let us create a user EXAMPLES (with CONNECT and RESOURCE roles) in pluggable database XEPDB1 in Oracle Database 21c Express Edition Release 21.0.0.0.0 - Production Version 21.3.0.0.0 (using the SYSTEM account in XEPDB1) and let us connect to this schema from SQL* Plus (see Listing 11).

Listing 11. Creating the EXAMPLE user

```
conn system@XEPDB1

CREATE USER "EXAMPLES" IDENTIFIED BY "examples"
DEFAULT TABLESPACE "USERS"
TEMPORARY TABLESPACE "TEMP";
ALTER USER "EXAMPLES" QUOTA 20M ON "USERS";
GRANT "CONNECT", "RESOURCE" TO "EXAMPLES" ;

conn EXAMPLES@XEPDB1
```

Let us create an analogous table CITIES in this schema, using native Oracle SDO_GEOMETRY type (and without auto-incrementation, that is, without Identity Column), see Listing 12.

Listing 12. Creating the CITIES table in Oracle

```
CREATE TABLE cities(
id NUMBER CONSTRAINT cities_PK PRIMARY KEY,
name VARCHAR2(50 CHAR) NOT NULL,
population NUMBER NOT NULL,
location SDO_GEOMETRY NOT NULL
);
```

To import the data from MySQL DBMS to Oracle XE database, let us execute in MySQL a properly prepared query which gives as a result the appropriate insert statements that can later be used in Oracle, see Listing 13. This query takes advantage of MySQL `CONCAT` function that concatenates given texts, MySQL `ST_AsText` function that returns a WKT representation of a given geometry and an Oracle constructor of a SDO_GEOMETRY type taking a WKT and a SRID as input arguments. Let us notice that this export-import idea can be used thanks to the WKT representation and thanks to an optional argument of `ST_AsText` function in MySQL that allows to establish axis order in that representation. For in Oracle's version of the WKT representation longitude goes first, before latitude, which makes it impossible to just take a copy of a default WKT representation from MySQL.

Listing 13. Creating the insert statements to enable the export of data from MySQL to Oracle XE (using ST_AsText function)

```
-- in EXAMPLES database in MYSQL DBMS:

SELECT CONCAT(
   "INSERT INTO cities VALUES(", c.id,",'",c.name,"',",c.population, ",
    SDO_GEOMETRY('", ST_AsText(c.location, 'axis-order=long-lat'),"',
     4326));") TEXT
FROM cities c;

+----------------------------------------------------------------------+
| TEXT                                                                 |
+----------------------------------------------------------------------+
| INSERT INTO cities VALUES
  (1,'Warszawa',1860281,SDO_GEOMETRY('POINT(21.02 52.12)', 4326)); |
| INSERT INTO cities VALUES
  (2,'Kraków',800653,SDO_GEOMETRY('POINT(19.57 50.03)', 4326));    ||
  INSERT INTO cities VALUES
  (3,'Wrocław',672929,SDO_GEOMETRY('POINT(17.02  51.07)',  4326));  |
|  INSERT  INTO  cities  VALUES
  (4,'Łódź',670642,SDO_GEOMETRY('POINT(19.28 51.47)',4326));          |
|  INSERT  INTO  cities  VALUES
  (5,'Poznań',546859,SDO_GEOMETRY('POINT(16.55  52.25)',  4326));   |
|  INSERT  INTO  cities  VALUES
  (6,'Gdańsk',486022,SDO_GEOMETRY('POINT(18.38  54.22)',  4326));   |
|  INSERT  INTO  cities  VALUES
  (7,'Szczecin',396168,SDO_GEOMETRY('POINT(14.34 53.26)', 4326));  |
| INSERT INTO cities VALUES
  (8,'Bydgoszcz',337666,SDO_GEOMETRY('POINT(18 53.07)', 4326));    |
| INSERT INTO cities VALUES
  (9,'Lublin',334681,SDO_GEOMETRY('POINT(22.34 51.14)', 4326));    |
| INSERT INTO cities VALUES
  (10,'Białystok',294242,SDO_GEOMETRY('POINT(23.1 53.08)', 4326)); |
+----------------------------------------------------------------------+
10 rows in set (0.00 sec)
```

By executing obtained insert statements on an EXAMPLE user account, we populate the table CITIES with appropriate data. We can easily check that the geometries created this way are valid (see Listing 14).

Listing 14. Validation of the geometries from the table CITIES

```
SELECT c.id, c.location.ST_IsValid()
FROM cities c
ORDER BY c.id ASC;

        ID C.LOCATION.ST_ISVALID()
---------- ------------------------
         1                        1
         2                        1
         3                        1
         4                        1
         5                        1
         6                        1
         7                        1
         8                        1
         9                        1
        10                        1
10 rows selected.
```

In Oracle Spatial and Graph 21c, we have to take care of introducing the metadata on our own executing an appropriate insert statement to USER_SDO_GEOM_METADATA view, see Listing 15 (0.5 stands there for a tolerance value for each of the dimensions).

Listing 15. Inserting the metadata for the table CITIES

```
INSERT INTO user_sdo_geom_metadata VALUES (
'CITIES',
'LOCATION',
```

```
SDO_DIM_ARRAY(
SDO_DIM_ELEMENT('longitude',-180,180,0.5),
SDO_DIM_ELEMENT('latitude',-90,90,0.5)
),
4326
);
COMMIT;
```

Asking the database to display the content of the table CITIES, we are able to see the native version of Oracle's constructor of a SDO_GEOMETRY type, see Listing 16.

LISTING 16. The contents of table CITIES in Oracle XE database (with native constructors calls)

```
SET PAGESIZE 50
SELECT * FROM cities c ORDER BY c.population DESC;

ID NAME     POPULATION  LOCATION(SDO_GTYPE,SDO_SRID,SDO_POINT(X,Y,Z),SDO_ELEM_INFO,SDO_ORDINATES)
- ------ -------- ----------------------------------------------------------
1 Warszawa 1860281 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(21.02, 52.12, NULL), NULL, NULL)
2 Kraków    800653 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(19.57, 50.03, NULL), NULL, NULL)
3 Wrocław   672929 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(17.02, 51.07, NULL), NULL, NULL)
4 Łódź      670642 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(19.28, 51.47, NULL), NULL, NULL)
5 Poznań    546859 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(16.55, 52.25, NULL), NULL, NULL)
6 Gdańsk    486022 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(18.38, 54.22, NULL), NULL, NULL)
7 Szczecin  396168 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(14.34, 53.26, NULL), NULL, NULL)
8 Bydgoszcz  337666 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(18, 53.07, NULL), NULL, NULL)
9 Lublin     334681 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(22.34, 51.14, NULL), NULL, NULL)
10 Białystok 294242 SDO_GEOMETRY(2001, 4326, SDO_POINT_TYPE(23.1, 53.08, NULL), NULL, NULL)
10 rows selected.
```

To display the WKT and WKB representations of the cities (here for instance for the cities with names ending with the letter "w"), one can use `Get_WKT()` and `Get_WKB()` methods called on objects stored in LOCATION column or appropriate functions from SDO_UTIL package, see Listing 17.

LISTING 17. The WKT and WKB representations of locations of the cities with names ending with letter "w"

```
SELECT c.name NAME, c.location.Get_WKT() WKT, c.location.Get_WKB() WKB
FROM cities c
WHERE SUBSTR(c.name,-1,1)='w'
ORDER BY c.population DESC;

SELECT c.name NAME, SDO_UTIL.TO_WKTGEOMETRY(c.location) WKT,
       SDO_UTIL.TO_WKBGEOMETRY(c.location) WKB
FROM cities c
WHERE SUBSTR(c.name,-1,1)='w'
ORDER BY c.population DESC;

NAME      WKT                   WKB
--------- --------------------- ----------------------------------------
Kraków    POINT  (19.57  50.03)  0000000001403391EB851EB852404903D70A3D70A4
Wrocław   POINT (17.02 51.07)   0000000001403105 1EB851EB85404988F5C28F5C29
```

Similarly the GML and GeoJSON representation can be obtained (here for the city of Łódź), see Listing 18.

LISTING 18. The GML and GeoJSON representations of location of the city of Łódź

```
SET LONG 40000

SELECT SDO_UTIL.TO_GMLGEOMETRY (c.location) GML_v2
FROM cities c
WHERE c.name='Łódź';

GML_V2
--------------------------------------------------------------------------------
<gml:Point srsName="EPSG:4326" xmlns:gml="http://www.opengis.net/gml">
<gml:coordinates decimal="." cs="," ts=" ">19.28,51.47 </gml:coordinates>
</gml:Point>
```

```
SELECT SDO_UTIL.TO_GML311GEOMETRY (c.location) GML_v3
FROM cities c
WHERE c.name='Łódź';

GML_V3
--------------------------------------------------------------------------------
<gml:Point srsName="EPSG:4326" xmlns:gml="http://www.opengis.net/gml">
<gml:pos srsDimension="2">19.28 51.47 </gml:pos></gml:Point>

SELECT SDO_UTIL.TO_GEOJSON(c.location) GEOJSON
FROM cities c
WHERE c.name='Łódź';

GEOJSON
--------------------------------------------------------------------------------
{ "type": "Point", "coordinates": [19.28, 51.47] }
```

It is worth mentioning that in `Get_GML` method we can change the coordinates order in GML format, see Listing 19.

LISTING 19. The GML representations of location of the city of Łódź with changed order of the coordinates

```
SELECT c.location.GET_GML() "GML long/lat",
       c.location.GET_GML(1) "GML lat/long"
FROM cities c
WHERE c.name='Łódź';

GML long/lat                                GML lat/long
------------------------------------------- -------------------------------
<gml:Point srsName="EPSG:4326"              <gml:Point srsName="EPSG:4326"
xmlns:gml=                                  xmlns:gml=
"http://www.opengis.net/gml">               "http://www.opengis.net/gml">
<gml:coordinates decimal="."                <gml:coordinates decimal="."
cs="," ts=" ">19.28,51.47                   cs="," ts=" ">51.47,19.28
</gml:coordinates></gml:Point>              </gml:coordinates></gml:Point>
```

Let us also notice that the similar export-import idea as presented earlier could involve Oracle `SDO_UTIL.FROM_GEOJSON` function and MySQL `ST_AsGeoJSON` function (without any problems with coordination's order), see Listing 20. One could also involve `SDO_UTIL.FROM_WKTGEOMETRY` function (together with MySQL `ST_AsText` function, with the change of the coordinates order), but in that case we obtain geometries with NULL value as a SRID (as the WKT representation itself does not contain the information about the coordinate system used), so after the execution of insert statement in Oracle XE, it is necessary to update all imported rows setting SDO_SRID field value to the appropriate coordinate system identifier, see Listing 21.

LISTING 20. Creating the insert statements to enable the export of data from MySQL to Oracle XE (using `SDO_UTIL.FROM_GEOJSON` function)

```
-- in EXAMPLES database in MYSQL DBMS:

SELECT CONCAT("INSERT INTO cities VALUES(", c.id,",'",c.name,"',",c.population, ",  SDO_UTIL.
    FROM_GEOJSON('", ST_AsGeoJSON(c.location), "', NULL, 4326));") TEXT
FROM cities c;

+--------------------------------------------------------------------------------+
| TEXT                                                                           |
+--------------------------------------------------------------------------------+
| INSERT INTO cities VALUES(1,'Warszawa',1860281,
  SDO_UTIL.FROM_GEOJSON
('{"type": "Point", "coordinates": [21.02, 52.12]}',  NULL, 4326));    ||
INSERT INTO cities VALUES(2,'Kraków',800653,
  SDO_UTIL.FROM_GEOJSON
  ('{"type": "Point", "coordinates": [19.57, 50.03]}',  NULL, 4326));    |
...
+--------------------------------------------------------------------------------+
10 rows in set (0.00 sec)
```

Listing 21. Creating the insert statements to enable the export of data from MySQL to Oracle XE (using SDO_UTIL.FROM_WKTGEOMETRY function and updating a SRID identifier after import)

```
-- in EXAMPLES database in MYSQL DBMS:

SELECT CONCAT("INSERT INTO cities VALUES(", c.id,",'",c.name,"',",c.population, ",SDO_UTIL.
    FROM_WKTGEOMETRY('", ST_AsText(c.location,  'axis-order=long-lat'),"'));") TEXT
FROM cities c;

+------------------------------------------------------------------------+
| TEXT                                                                   |
+------------------------------------------------------------------------+
| INSERT INTO cities VALUES(1,'Warszawa',1860281,
  SDO_UTIL.FROM_WKTGEOMETRY('POINT(21.02 52.12)'));|                     |
  INSERT INTO cities VALUES(2,'Kraków',800653,
  SDO_UTIL.FROM_WKTGEOMETRY('POINT(19.57 50.03)'));                      |
...

+------------------------------------------------------------------------+
10 rows in set (0.00 sec)

-- in Oracle XE database, after insert statements:

UPDATE cities c
SET c.location.SDO_SRID=4326;
COMMIT;
```

Summarizing these export-import ideas, it can be seen that in the case of these two systems, using the GeoJSON representation seems to be the most elegant and the least troublesome.

In Oracle database, we can also export geometries to JSON, KML and GeoHash, see Listing 22.

Listing 22. JSON, KML and GeoHash representations of locations of chosen cities

```
SELECT c.name NAME, SDO_UTIL.TO_JSON_JSON(c.location) JSON
FROM cities c
WHERE c.population BETWEEN 500000 AND 600000
ORDER BY c.population DESC;

NAME     JSON
-------- ------------------------------------------------------------------
Poznań   {"srid":4326,"point":{"directposition":[16.55,52.25]}}

SELECT c.name NAME, SDO_UTIL.TO_KMLGEOMETRY(c.location) KML
FROM cities c
WHERE c.name LIKE 'By%';

NAME         KML
------------ ------------------------------------------------------------
Bydgoszcz    <Point><extrude>0</extrude><tessellate>0</tessellate>
             <altitudeMode>relativeToGround</altitudeMode>
             <coordinates>18.0,53.07 </coordinates></Point>

SELECT c.name NAME, SDO_CS.TO_GEOHASH(c.location, 15) GEOHASH
FROM cities c
WHERE c.name LIKE 'K%';

NAME       GEOHASH
-------- ----------------------------------------------------------------
Kraków     u2vut7dmy5cwszc
```

We also have the ability to validate WKT or WKB representations (with OGC definitions), see Listing 23 with WKT validation.

Listing 23. Validation of a chosen WKT representation

```
SELECT SDO_UTIL.VALIDATE_WKTGEOMETRY('POINT(21.02 52.12)') VALIDATION
FROM DUAL;
```

```
VALIDATION
--------------------------------------------------------------------------
TRUE
```

Listing 24 demonstrates how to invoke sample functions or methods for elements of ST_GEOMETRY type. To obtain elements of this type we cast values of LOCATION column of the CITIES table to ST_GEOMETRY.

Listing 24. The chosen functions and methods invoked for ST_GEOMETRY objects

```
SELECT ST_GEOMETRY(c.location).GET_WKT() WKT
FROM cities c
WHERE c.name='Lublin';

WKT
--------------------------------------------------------------------------
POINT (22.34 51.14)

SELECT ST_GEOMETRY.ST_ASTEXT(ST_GEOMETRY(c.location)) WKT
FROM cities c
WHERE c.name='Poznań';

WKT
--------------------------------------------------------------------------
POINT (16.55 52.25)

SELECT mdsys.ogc_ASTEXT(ST_GEOMETRY(c.location)) WKT
FROM cities c
WHERE c.name='Gdańsk';

WKT
--------------------------------------------------------------------------
POINT (18.38 54.22)
```

Let us now move (using default postgresql user) to the database postgis_34_sample in PostgreSQL 14 with PostGIS version 3.4.1 and create there the table cities by executing via SQL Shell (psql) a statement shown in Listing 25. Let us notice that in practice, when creating such a table, we probably would assign the location column not the geometry type but the geography type. In this paper, we have chosen geometry to be able to show how certain functions work.

Listing 25. Creating the cities table in PostgreSQL/ PostGIS

```
CREATE TABLE cities(
id SERIAL CONSTRAINT cities_PK PRIMARY KEY,
name VARCHAR(50) NOT NULL,
population INT NOT NULL,
location GEOMETRY(POINT, 4326));
```

The appropriate layer metadata in PostGIS are filled automatically; it can be checked by executing the query from Listing 26.

Listing 26. Checking the metadata for the table cities

```
SELECT *
FROM GEOMETRY_COLUMNS
WHERE f_table_name='cities';

f_table_catalog  |f_table_schema|f_table_name|f_geometry_column|coord_dimension| srid | type
-----------------+--------------+------------+-----------------+---------------+------+------
postgis_34_sample| public       | cities     | location        |             2 | 4326 | POINT
(1 row)
```

In PostGIS, as in Oracle, in the WKT representation the longitude goes first. Let us come back to the Oracle database again, and prepare and execute there a query that enables export of data to the postgis_34_sample database, see Listing 27. That query uses an Oracle concatenation operator ||, the possibility to access the coordinates of a point through the fields of the SDO_POINT object (which is a field of SDO_GEOMETRY object), and `ST_PointFromText` function in PostGIS module.

LISTING 27. Creating the insert statements to enable the export of data from Oracle XE to postgis_34_sample database (using ST_PointFromText function)

```
-- in Oracle XE database:

SELECT 'INSERT INTO cities  VALUES(' || c.id ||  ', ''' || c.name ||''', ' || c.population || ',
    ST_PointFromText(''POINT(' || c.location.sdo_point.x || ' ' ||  c.location.sdo_point.y || ')''
    ,4326));' TEXT
FROM cities c;

TEXT
--------------------------------------------------------------------------
INSERT INTO cities  VALUES(1, 'Warszawa', 1860281, ST_PointFromText('POINT(21.02 52.12)',4326));
INSERT INTO cities  VALUES(2, 'Kraków', 800653, ST_PointFromText('POINT(19.57 50.03)',4326));
INSERT INTO cities  VALUES(3, 'Wrocław', 672929, ST_PointFromText('POINT(17.02 51.07)',4326));
INSERT INTO cities  VALUES(4, 'Łódź', 670642, ST_PointFromText('POINT(19.28 51.47)',4326));
INSERT INTO cities  VALUES(5, 'Poznań', 546859, ST_PointFromText('POINT(16.55 52.25)',4326));
INSERT INTO cities  VALUES(6, 'Gdańsk', 486022, ST_PointFromText('POINT(18.38 54.22)',4326));
INSERT INTO cities  VALUES(7, 'Szczecin', 396168, ST_PointFromText('POINT(14.34 53.26)',4326));
INSERT INTO cities  VALUES(8, 'Bydgoszcz', 337666, ST_PointFromText('POINT(18 53.07)',4326));
INSERT INTO cities  VALUES(9, 'Lublin', 334681, ST_PointFromText('POINT(22.34 51.14)',4326));
INSERT INTO cities  VALUES(10, 'Białystok', 294242, ST_PointFromText('POINT(23.1 53.08)',4326));
10 rows selected.
```

Let us change the client encoding in SQL Shell to WIN1251 (to solve the problem with Polish diacritics) by executing the command presented in Listing 28.

LISTING 28. Encoding change

```
SET client_encoding = 'WIN1251';
```

Now the insert statements obtained in Oracle can be executed in postgis_34_sample database. After the import we can check that all geometries are valid ones, see Listing 29.

LISTING 29. Validation of the geometries from the table cities

```
SELECT c.id, ST_IsValid(c.location) FROM cities c ORDER BY c.id;

 id | st_isvalid
----+------------
  1 | t
  2 | t
  3 | t
  4 | t
  5 | t
  6 | t
  7 | t
  8 | t
  9 | t
 10 | t
(10 rows)
```

Let us now choose the cities that are within 200 kilometres from city of Warszawa, and display their names, along with their WKT and EWKT representations, see Listing 30. Let us notice that before filtering the cities located within some distance from the capital of Poland, we cast the geometry type to geography.

LISTING 30. Selecting cities located within 200 kilometres of the city of Warszawa

```
SELECT c.name "NAME", ST_AsText(c.location) "WKT",
       ST_AsEWKT(c.location) "EWKT"
FROM cities c
WHERE ST_DWithin(c.location:: geography, (SELECT d.location
                                          FROM CITIES d
                                          WHERE d.name='Warszawa')::geography, 200000)
ORDER BY c.id;

   NAME    |        WKT         |             EWKT
-----------+--------------------+------------------------------
 Warszawa  | POINT(21.02 52.12) | SRID=4326;POINT(21.02 52.12)
```

```
 Łódź       | POINT(19.28 51.47) | SRID=4326;POINT(19.28 51.47)
 Lublin     | POINT(22.34 51.14) | SRID=4326;POINT(22.34 51.14)
 Białystok | POINT(23.1 53.08)  | SRID=4326;POINT(23.1 53.08)
(4 rows)
```

We can also covert the geometries to WKB, EWKB, HEXEWKB, TWKB formats, see Listing 31, and to GML, GeoJSON, GeoHash, KML and SVG, see Listing 32, or just put many functions together, see Listing 33.

Listing 31. Displaying the WKB, EWKB, HEXEWKB and TWKB representations of the locations of the selected cities

```
SELECT ST_AsBinary(c.location) "WKB", ST_AsTWKB(c.location) "TWKB"
FROM cities c
WHERE c.id=5;

                      WKB                       |           TWKB
------------------------------------------------+------------------------
 \x0101000000cdccccccccc8c3040000000000000204a40 | \xa100b083ca01d0e8fd04
(1 row)

SELECT ST_AsEWKB(c.location) "EWKB"
FROM cities c
WHERE c.id=5;

                          EWKB
---------------------------------------------------------
 \x0101000020e6100000cdccccccccc8c3040000000000000204a40
(1 row)

SELECT c.name "NAME", ST_AsHEXEWKB(c.location) "HEXEWKB"
FROM cities c
WHERE c.name LIKE '_r%';

  NAME    |                         HEXEWKB ---------
+-----------------------------------------------------Kraków   |
0101000020E610000052B81E85EB913340A4703D0AD7034940
 Wrocław | 0101000020E610000085EB51B81E053140295C8FC2F5884940 (2 rows)
```

Listing 32. Displaying the GML, GeoJSON, GeoHash, KML and SVG representations of the location of the city of Łódź

```
SELECT ST_AsGML(c.location) "GML"
FROM cities c
WHERE c.name='Łódź';
                                    GML
------------------------------------------------------------------------
 <gml:Point srsName="EPSG:4326"><gml:coordinates>19.28,51.47</gml:coordinates>
 </gml:Point>
(1 row)

SELECT ST_AsGeoJSON(c.location) "GeoJSON",
       ST_GeoHash(c.location) "GeoHash"
FROM cities c
WHERE c.name='Łódź';

                     GeoJSON                     |       GeoHash
-------------------------------------------------+----------------------
 {"type":"Point","coordinates":[19.28,51.47]} | u3jsy9z62x3vr6bkdewj
(1 row)

SELECT ST_AsKML(c.location) "KML", ST_AsSVG(c.location) "SVG"
FROM cities c
WHERE c.name='Łódź';


                    KML                    |           SVG
-------------------------------------------+--------------------------
```

```
 <Point><coordinates>19.28,51.47        | cx="19.28" cy="-51.47"
 </coordinates></Point>
(1 row)
```

LISTING 33. Displaying the WKT representation of the union of geometries representing two selected cities (with additional use of functions converting geometry to and from WKB representation)

```
SELECT ST_AsText(
       ST_MPointFromWKB(ST_AsBinary(ST_UNION(c.location)))) "MULTIPOINT"
FROM cities c
WHERE c.id IN (1,2);

              MULTIPOINT
---------------------------------------
 MULTIPOINT((19.57 50.03),(21.02 52.12))
(1 row)
```

5. Conclusion

Spatial relation and object-relational database management systems use various formats to represent the geometric objects: both those defined and required by the main international standards (such as WKT, WKB and GML formats), those originally built for some application and then adopted as an international standard (such as KML format), those that are not maintained by any standards organization (such as GeoJSON) and those based on some algorithmic ideas (as Geohash). Some of the systems also propose and use their own extensions of such formats (e.g. PostGIS/PostgreSQL with EWKT and EWKB formats). Implementing functions and methods to support the handling of these formats through different relational and object-relational systems definitely facilitates work in these systems and the export and import of the spatial data. The birth of new lightweight formats used widely in the NoSQL databases (such as JSON/GeoJSON) forced the traditional database management systems to adopt these formats which expanded the capabilities of these systems and gives hope for some interoperability between traditional and new types of spatial databases management systems.

Funding No funding.
Ethical approval Not applicable.
Data availability The data used and analyzed in the paper is openly available on websites:

- Współrzędne geograficzne polskich miast (Geographic coordinates of Polish cities), `https://astronomia.zagan.pl/art/wspolrzedne.html`, author: Patka, J.
- Informacja o wynikach Narodowego Spisu Powszechnego Ludności i Mieszkań 2021 na poziomie województw, powiatów i gmin (Information on the results of the National of Population and Housing Census 2021 at the level voivodeships, poviats and communes), `https://stat.gov.pl/download/gfx/portalinformacyjny/pl/defaultaktualnosci/6536/1/1/1/informacja_o_wynikach_narodowego_spisu_powszechnego_ludnosci_i_mieszkan_2021.pdf`, Główny Urząd Statystyczny (Statistics Poland).

References

[1] GeoJSON. `https://geojson.org/`, 2019. Last Accessed May 7, 2024.
[2] Introducing JSON. `https://www.json.org/json-en.html`, 2024. Last Accessed May 7, 2024.
[3] S. Agarwal and K. Rajan. Performance analysis of MongoDB versus PostGIS/PostGreSQL databases for line intersection and point containment spatial queries. *Spatial Information Research*, 24:671–677, 2016.
[4] S. Agarwal and K. Rajan. Analyzing the performance of NoSQL vs. SQL databases for Spatial and Aggregate queries. In *Free and Open Source Software for Geospatial (FOSS4G) Conference Proceedings*, volume 17, 2017. Available from: `https://scholarworks.umass.edu/foss4g/vol17/iss1/4`.
[5] P. Amirian, A. C. Winstanley, and A. Basiri. NoSQL storage and management of geospatial data with emphasis on serving geospatial data using standard geospatial web services. In *GIS Research UK (GISRUK) 2013*, pages 1–5, 2013. Available from: `https://mural.maynoothuniversity.ie/6473/`.
[6] C. Baptista, O. Junior, M. Oliveira, F. Andrade, T. Silva, and C. S. Pires. Using OGC services to interoperate spatial data stored in SQL and NoSQL databases. pages 61–72, 2011.

[7] E. Baralis, A. D. Valle, P. Garza, C. Rossi, and F. Scullino. SQL versus NoSQL databases for geospatial applications. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 3388–3397, 2017. Available from: https://www.researchgate.net/publication/322516902_SQL_versus_NoSQL_databases_for_geospatial_applications, https://ieeexplore.ieee.org/document/8258324.
[8] D. Bartoszewski, A. Piórkowski, and M. Lupa. The Comparison of Processing Efficiency of Spatial Data for PostGIS and MongoDB Databases. In S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, editors, *Beyond Databases, Architectures and Structures. Paving the Road to Smart Data Processing and Analysis*, pages 291–302, Cham, 2019. Springer International Publishing.
[9] Butler, H. and Daly, M. and Doyle, A. and Gillies, S. and Hagen, S. and Schaub, T. RFC 7946: The GeoJSON Format. https://www.rfc-editor.org/rfc/rfc7946, 2016. Last Accessed May 7, 2024.
[10] Cover Pages. Geography Markup Language (GML). http://xml.coverpages.org/geographyML.html, 2008. Last Accessed May 7, 2024.
[11] DataStax. Indexing and querying polygons | CQL for DataStax Enterprise | DataStax Docs. https://docs.datastax.com/en/cql/dse/cycling-examples/demoGeo.html, 2024. Last Accessed May 7, 2024.
[12] Ecma International. ECMA-404, The JSON data interchange syntax, 2nd edition, December 2017. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/, 2017. Last Accessed May 7, 2024.
[13] W. B. Filho, H. V. Olivera, M. Holanda, and A. A. Favacho. Geographic Data Modeling for NoSQL Document-Oriented Databases. In *The Seventh International Conference on Advanced Geographic Information Systems, Applications, and Services, Lisboa, 2015*, pages 63–68, 2015. Available from: https://personales.upv.es/thinkmind/dl/conferences/geoprocessing/geoprocessing_2015/geoprocessing_2015_4_10_30113.pdf.
[14] Geographic Information Systems Stack Exchange. WKT coordinate inner order (long, lat). https://gis.stackexchange.com/questions/366853/wkt-coordinate-inner-order-long-lat, 2022. Last Accessed May 7, 2024.
[15] GitHub. TopoJSON. https://github.com/topojson/topojson/wiki, 2016. Last Accessed May 7, 2024.
[16] Główny Urząd Statystyczny (Statistics Poland). Informacja o wynikach Narodowego Spisu Powszechnego Ludności i Mieszkań 2021 na poziomie województw, powiatów i gmin (Information on the results of the National of Population and Housing Census 2021 at the level voivodeships, poviats and communes). https://stat.gov.pl/download/gfx/portalinformacyjny/pl/defaultaktualnosci/6536/1/1/1/informacja_o_wynikach_narodowego_spisu_powszechnego_ludnosci_i_mieszkan_2021.pdf, Sep 2022. Last Accessed July 4, 2023.
[17] Google Developers. Google Developers, Google Maps Platform: Encoded Polyline Algorithm Format. https://developers.google.com/maps/documentation/utilities/polylinealgorithm?hl=en, 2024. Last Accessed May 7, 2024.
[18] D. Guo and E. Onstein. State-of-the-Art Geospatial Information Processing in NoSQL Databases. *ISPRS International Journal of Geo-Information*, 9(5):331, 2020. Available from: https://www.mdpi.com/2220-9964/9/5/331/pdf?version=1590501181.
[19] J. A. Herrera-Ramírez, M. Treviño-Villalobos, and L. Víquez-Acuña. Hybrid storage engine for geospatial data using NoSQL and SQL paradigms. *Revista Tecnología en Marcha*, 34:40–54, 2021. Available from: http://www.scielo.sa.cr/scielo.php?script=sci_arttext&pid=S0379-39822021000100040&nrm=iso.
[20] ISO. ISO/IEC 2382-1:1993, Information technology - Vocabulary - Part 1: Fundamental terms. https://www.iso.org/standard/7229.html, 1993. Last Accessed May 7, 2024.
[21] ISO. ISO 19125-1:2004, Geographic information - Simple feature access - Part 1: Common architecture. https://www.iso.org/standard/40114.html, 2004. Last Accessed May 7, 2024.
[22] ISO. ISO 19125-2:2004, Geographic information - Simple feature access - Part 2: SQL option. https://www.iso.org/standard/40115.html, 2004. Last Accessed May 7, 2024.
[23] ISO. ISO 19136:2007, Geographic information - Geography Markup Language (GML). https://www.iso.org/standard/32554.html, 2007. Last Accessed May 7, 2024.
[24] ISO. ISO/IEC 13249-3:2016 Information technology - Database languages - SQL multimedia and application packages - Part 3: Spatial. https://www.iso.org/standard/60343.html, 2016. Last Accessed May 7, 2024.
[25] ISO. ISO 19136-1:2020, Geographic information - Geography Markup Language (GML) - Part 1: Fundamentals. https://www.iso.org/standard/75676.html, 2020. Last Accessed May 7, 2024.
[26] Krishnan, A. Hackr.io: JSON vs XML in 2024: Comparing Features and Examples. https://hackr.io/blog/json-vs-xml, 2024. Last Accessed May 7, 2024.
[27] D. Laksono. Testing Spatial Data Deliverance in SQL and NoSQL Database Using NodeJS Fullstack Web App. In *2018 4th International Conference on Science and Technology (ICST)*, pages 1–5, 2018.
[28] Y. Li, D. Kim, and B.-S. Shin. Geohashed Spatial Index Method for a Location-Aware WBAN Data Monitoring System Based on NoSQL. *Journal of Information Processing Systems*, 12(2):263–274, 2016. Available from: https://koreascience.kr/article/JAKO201621650894684.pdf.
[29] C.-T. Lu, R. F. D. S. Jr, L. N. Sripada, and Y. Kou. Advances in GML for Geospatial Applications. *Geoinformatica*, 11:131–157, 2007.
[30] Open Geospatial Consortium. OpenGIS Simple Features Implementation Specification for SQL, 1.1, 99-049. https://portal.ogc.org/files/?artifact_id=829, May 1999. Last Accessed May 7, 2024.
[31] Open Geospatial Consortium. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture, 1.1.0, 05-126. https://portal.ogc.org/files/?artifact_id=13227, Nov 2005. Last Accessed May 7, 2024.
[32] Open Geospatial Consortium. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option, 1.1.0, 05-134. https://portal.ogc.org/files/?artifact_id=13228, Nov 2005. Last Accessed May 7, 2024.
[33] Open Geospatial Consortium. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture, 1.2.0, 06-103r3. https://portal.ogc.org/files/?artifact_id=18241, Oct 2006. Last Accessed May 7, 2024.
[34] Open Geospatial Consortium. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option, 1.2.0, 06-104r3. https://portal.ogc.org/files/?artifact_id=18242, Oct 2006. Last Accessed May 7, 2024.
[35] Open Geospatial Consortium. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option, 1.2.1, 06-104r4. https://portal.ogc.org/files/?artifact_id=25354, Aug 2010. Last Accessed May 7, 2024.
[36] Open Geospatial Consortium. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture, 1.2.1, 06-103r4. https://portal.ogc.org/files/?artifact_id=25355, May 2011. Last Accessed May 7, 2024.

[37] Open Geospatial Consortium. Certified Product List (Product Provider: Crunchy Data). https://www.ogc.org/resources/certified-products/?display_opt=1&org_match=Crunchy%20Data, 2023. Last Accessed July 4, 2023.
[38] Open Geospatial Consortium. Certified Product List (Product Provider: Oracle USA). https://www.ogc.org/resources/certified-products/?display_opt=1&org_match=Oracle%20USA, 2023. Last Accessed July 4, 2023.
[39] Open Geospatial Consortium. Geography Markup Language. https://www.ogc.org/standards/gml, 2023. Last Accessed May 7, 2024.
[40] Open Geospatial Consortium. KML. https://www.ogc.org/standards/kml, 2023. Last Accessed May 7, 2024.
[41] Open Geospatial Consortium. Simple Feature Access - Part 1: Common Architecture. https://www.ogc.org/standards/sfa, 2023. Last Accessed May 7, 2024.
[42] Open Geospatial Consortium. Simple Feature Access - Part 2: SQL Option. https://www.ogc.org/standards/sfs, 2023. Last Accessed May 7, 2024.
[43] Oracle. Oracle Database Online Documentation 12c Release 1 (12.1) E41152-15, Application Development, XML DB Developer's Guide, 39.1.2: Overview of JSON Compared with XML. https://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6250, Dec 2016. Last Accessed July 4, 2023.
[44] Oracle. Spatial and Graph Analytics with Oracle Database 12c Release 2, ORACLE WHITEPAPER, NOVEMBER 2016. https://www.oracle.com/technetwork/database/options/spatialandgraph/spatial-and-graph-wp-12c-1896143.pdf, Nov 2016. Last Accessed May 7, 2024.
[45] Oracle. Spatial and Graph Analytics with Oracle Database 18, ORACLE WHITE PAPER, FEBRUARY 2018. https://download.oracle.com/otndocs/products/spatial/pdf/Spatial_and_Graph_TWP_18c.pdf, Feb 2018. Last Accessed May 7, 2024.
[46] Oracle. Oracle Spatial and Graph in Oracle Database 19c. https://www.oracle.com/a/tech/docs/spatial-and-graph-in-oracle-database-19c-preso.pdf, 2019. Last Accessed May 7, 2024.
[47] Oracle. MySQL 8.0 Reference Manual Including MySQL NDB Cluster 8.0, 2023-07-04, revision: 76058. https://downloads.mysql.com/docs/refman-8.0-en.a4.pdf, Jul 2023. Last Accessed July 4, 2023.
[48] Oracle. Oracle: Spatial and Graph features in Oracle Database. https://www.oracle.com/database/technologies/spatialandgraph.html, 2023. Last Accessed July 4, 2023.
[49] Oracle. Oracle ®Spatial Spatial Developer's Guide 21c F32277-17, July 2023. https://docs.oracle.com/en/database/oracle/oracle-database/21/spatl/spatial-developers-guide.pdf, Jul 2023. Last Accessed July 4, 2023.
[50] OSGeo. Axis Order Confusion. https://wiki.osgeo.org/wiki/Axis_Order_Confusion, 2018. Last Accessed May 7, 2024.
[51] Patka, J. Współrzędne geograficzne polskich miast (Geographic coordinates of Polish cities). https://astronomia.zagan.pl/art/wspolrzedne.html, 2008. Last Accessed May 7, 2024.
[52] PostGIS. PostGIS 2.2.1 Manual SVN Revision (14555). http://postgis.net/stuff/postgis-2.2.1.pdf, 2016. Last Accessed July 4, 2022.
[53] PostGIS. About PostGIS. https://postgis.net/, 2023. Last Accessed July 4, 2023.
[54] PostGIS. PostGIS 3.2.6dev Manual DEV (Thu 29 Jun 2023 04:28:52 AM UTC rev. 47ccf6a). https://postgis.net/stuff/postgis-3.2.pdf, Jun 2023. Last Accessed July 3, 2023.
[55] PostGIS. PostGIS 3.4.0dev Manual DEV (Wed 05 Jul 2023 08:34:57 AM UTC rev. 54f1530). https://postgis.net/stuff/postgis-3.4.pdf, Jul 2023. Last Accessed May 7, 2024.
[56] Ryeng, N. H. MySQL Blog Archive: Geography in MySQL 8.0. https://dev.mysql.com/blog-archive/geography-in-mysql-8-0/, 2018. Last Accessed May 7, 2024.
[57] Ryeng, N. H. MySQL Blog Archive: Upgrading to MySQL 8.0 with Spatial Data. https://dev.mysql.com/blog-archive/upgrading-to-mysql-8-0-with-spatial-data/, 2018. Last Accessed May 7, 2024.
[58] Safris, S. Toptal Engineering Blog: A Deep Look At JSON vs. XML, Part 2: The Strengths and Weaknesses of Both. https://www.toptal.com/web/json-vs-xml-part-2, 2019. Last Accessed May 7, 2024.
[59] P. O. Santos, M. M. Moro, and C. A. D. Jr. Comparative Performance Evaluation of Relational and NoSQL Databases for Spatial and Mobile Applications. In Q. Chen, A. Hameurlain, F. Toumani, R. Wagner, and H. Decker, editors, *Database and Expert Systems Applications*, pages 186–200, Cham, 2015. Springer International Publishing.
[60] solidIT consulting & software development gmbh. DB-Engines Ranking of Relational DBMS. https://db-engines.com/en/ranking/relational+dbms, 2024. Last Accessed May 7, 2024.
[61] I. S. Suwardi, D. Dharma, D. P. Satya, and D. P. Lestari. Geohash index based spatial data model for corporate. In *2015 International Conference on Electrical Engineering and Informatics (ICEEI)*, pages 478–483, 2015. Available from: https://ieeexplore.ieee.org/document/7352548.
[62] The PostgreSQL Global Development Group. The PostgreSQL Global Development Group, PostgreSQL 14.8 Documentation. https://www.postgresql.org/files/documentation/pdf/14/postgresql-14-A4.pdf, 2023. Last Accessed July 4, 2023.
[63] Veness, C. Movable Type Scripts, Geohashes: Geohash encoding/decoding. https://www.movable-type.co.uk/scripts/geohash.html, 2022. Last Accessed May 7, 2024.
[64] Whelan, P. Geohash Intro - Big Fast Blog. https://bigfastblog.com/geohash-intro, 2011. Last Accessed May 7, 2024.
[65] M. Wyszomirski. Przegląd możliwości zastosowania wybranych baz danych NoSQL do zarządzania danymi przestrzennymi (An overview of possibilities of using selected NoSQL databases to manage spatial data). *ROCZNIKI GEOMATYKI*, 16:55–69, 2018. Available from: http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-cc5a0a99-1da7-4fd9-8a87-3d7a8226ab27/c/5_Wyszomirski_Przeglad_55-69.pdf.
[66] X. Zhang, W. Song, and L. Liu. An implementation approach to store GIS spatial data on NoSQL database. In *22nd International Conference on Geoinformatics, 2014*, 2014.

Faculty of Mathematics and Computer Science, University of Łódź
S. Banacha 22, 90-238 Łódź, Poland

Email address, Aleksandra Stasiak: aleksandra.stasiak@wmii.uni.lodz.pl

TABLE 1. Functions related to WKT and WKB formats described in SF standard - required functionality (based on [30])

Format	Function Name	Input Arguments	Output
WKT	AsText	a geometry value as a Geometry (or any subtype of this type)	the WKT representation of the given geometry (as a String)
	GeomFromText	a WKT representation of a geometry (as a String) and a SRID (as an Integer)	the geometry value for that WKT representation (as a Geometry)
	PointFromText, LineFromText, PolyFromText, MPointFromText, MLineFromText, MPolyFromText, GeomCollFromTxt	a WKT representation of a geometry (as a String) and a SRID (as an Integer)	the geometry value of an appropriate type (respectively: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection type) for that WKT representation
WKB	AsBinary	a geometry value as a Geometry (or any subtype of this type)	the WKB representation of the given geometry (as a Binary)
	GeomFromWKB	a WKB representation of a geometry (as Binary) and a SRID (as an Integer)	the geometry value for that WKB representation (as a Geometry)
	PointFromWKB, LineFromWKB, PolyFromWKB, MPointFromWKB, MLineFromWKB, MPolyFromWKB, GeomCollFromWKB	a WKB representation of a geometry (as a Binary) and a SRID (as an Integer)	the geometry value of an appropriate type (respectively: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection type) for that WKB representation

TABLE 2. Functions related to WKT and WKB formats described in SF standard - optional functionality (based on [30])

Format	Function Name	Input Arguments	Output
WKT	BdPolyFromText, BdMPolyFromText	a WKT representation of a collection of closed linestrings (as a String with representation of MultiLineString) and a SRID (as an Integer)	a Polygon or MultiPolygon, respectively, for that WKT representation
WKB	BdPolyFromWKB, BdMPolyFromWKB	a WKB representation of a collection of closed linestrings (as a binary representation of MultiLineString) and a SRID (as an Integer)	a Polygon or MultiPolygon, respectively, for that WKB representation

TABLE 3. Functions related to WKT and WKB formats provided by MySQL 8.0 (based on section 12.16 of [47])

Format	Function Name	Input Arguments	Output
WKT	ST_AsText and ST_AsWKT	a geometry value in internal format, optionally an additional argument describing the axis order, in case of geographic coordinates	the WKT representation of the given geometry
	ST_GeomFromText and ST_GeometryFromText	a WKT representation of a geometry of any type, optionally a SRID, optionally a parameter describing the axis order, in case of geographic coordinates	the geometry value for that WKT representation
	ST_PointFromText, ST_LineFromText and ST_LineStringFromText, ST_PolyFromText and ST_PolygonFromText, ST_MPointFromText and ST_MultiPointFromText, ST_MLineFromText and ST_MultiLineStringFromText, ST_MPolyFromText and ST_MultiPolygonFromText, ST_GeomCollFromText and ST_GeometryCollectionFromText and ST_GeomCollFromTxt	a WKT representation of a geometry of an appropriate type, optionally a SRID, optionally a parameter describing the axis order, in case of geographic coordinates	the geometry value of the appropriate type (i.e. a Point, a LineString, a Polygon, a MultiPoint, a MultiLineString, a MultiPolygon, a GeometryCollection, respectively) for that WKT representation
WKB	ST_AsBinary and ST_AsWKB	a geometry value in internal format, optionally an additional argument describing the axis order, in case of geographic coordinates	the WKB representation of the given geometry
	ST_GeomFromWKB and ST_GeometryFromWKB	a BLOB with a WKB representation of a geometry of any type, optionally a SRID, optionally a parameter describing the axis order, in case of geographic coordinates	the geometry value for that WKB representation
	ST_PointFromWKB, ST_LineFromWKB and ST_LineStringFromWKB, ST_PolyFromWKB and ST_PolygonFromWKB, ST_MPointFromWKB and ST_MultiPointFromWKB, ST_MLineFromWKB and ST_MultiLineStringFromWKB, ST_MPolyFromWKB and ST_MultiPolygonFromWKB, ST_GeomCollFromWKB and ST_GeometryCollectionFromWKB	a WKB representation of a geometry of an appropriate type, optionally a SRID, optionally a parameter describing the axis order, in case of geographic coordinates	the geometry value of the appropriate type (i.e. a Point, a LineString, a Polygon, a MultiPoint, a MultiLineString, a MultiPolygon, a GeometryCollection, respectively) for that WKB representation

TABLE 4. Functions related to GeoJSON and GeoHash formats in MySQL 8.0 (based on section 12.16 of [47])

Format	Function Name	Input Arguments	Output
GeoJSON	ST_AsGeoJSON	a geometry value, optionally a maximal number of decimal digits for coordinates, and some optional parameter	the geometry value in a GeoJSON format
	ST_GeomFromGeoJSON	a string with a geometry in GeoJSON format and an optional parameter with some options and optionally a SRID	the geometry value for that GeoJSON representation (without support for Features and FeatureCollections, but with possibility to extract the geometry objects from them)
GeoHash	ST_GeoHash	longitude, latitude and a max_length parameter or a POINT value (with X and Y coordinates that are in the valid ranges for longitude and latitude, respectively) and max_length	the geohash string corresponding to the location indicated by longitude and latitude or by a POINT geometry (string no longer that a max_length characters)
	ST_PointFromGeoHash	a geohash string value and a SRID	the geometry value of a POINT type with coordinates indicating the longitude and the latitude of the location described by the given geohash value
	ST_LatFromGeoHash, ST_LongFromGeoHash	a geohash string value	the latitude (respectively: the longitude) of location described by a given geohash value (as a double-precision numbers)

TABLE 5. Functions related to WKT, WKB and GML formats from SDO_UTIL package in Oracle XE 21c (based on [49])

Format	Function Name	Input Arguments	Output
WKT	SDO_UTIL.TO_WKTGEOMETRY	a geometry value (as an SDO_GEOMETRY object)	the WKT representation for that geometry (as a CLOB)
	SDO_UTIL.FROM_WKTGEOMETRY	a WKT representation of a geometry (as a CLOB or as a VARCHAR2)	the geometry value for that WKT representation (as an SDO_GEOMETRY object)
WKB	SDO_UTIL.TO_WKBGEOMETRY	a geometry value (as an SDO_GEOMETRY object)	the WKB representation for that geometry (as a BLOB)
	SDO_UTIL.FROM_WKBGEOMETRY	a WKB representation of a geometry (as a BLOB)	the geometry value for that WKB representation (as an SDO_GEOMETRY object)
GML	SDO_UTIL.TO_GMLGEOMETRY	a geometry value (as an SDO_GEOMETRY object) or a geometry value and a coordOrder parameter (as a NUMBER, reserved for Oracle use)	the GML representation for that geometry (as a CLOB), with 2.0 version of GML
	SDO_UTIL.TO_GML311GEOMETRY	a geometry value (as an SDO_GEOMETRY object)	the GML representation for that geometry (as a CLOB), with 3.1.1 version of GML
	SDO_UTIL.FROM_GMLGEOMETRY	a geometry values in GML 2.0 version (as a CLOB or a VARCHAR2) and srsNamespace parameter (as a VARCHAR2, reserved for Oracle use, by default set to NULL)	the geometry value for that GML representation (as an SDO_GEOMETRY object)
	SDO_UTIL.FROM_GML311GEOMETRY	a geometry values in GML 3.1.1 version (as CLOB or as a VARCHAR2), srsNamespace (as a VARCHAR2) and / or coordOrder (as a NUMBER)	the geometry value for that GML representation (as an SDO_GEOMETRY object)

TABLE 6. Functions related to GeoJSON / JSON and KML formats from SDO_UTIL package in Oracle XE 21c (based on [49])

Format	Function Name	Input Arguments	Output
GeoJSON and JSON	SDO_UTIL.TO_GEOJSON	a geometry value (as an SDO_GEOMETRY object)	the geometry value in GEOJSON format (as a CLOB)
	SDO_UTIL. TO_GEOJSON_JSON	a geometry value (as an SDO_GEOMETRY object)	the geometry of type JSON in GeoJSON format
	SDO_UTIL.TO_JSON and SDO_UTIL. TO_JSON_VARCHAR	a geometry value (as an SDO_GEOMETRY object)	the JSON representation of that geometry (respectively as a CLOB or as a VARCHAR2)
	SDO_UTIL.TO_JSON_JSON	a geometry value (as an SDO_GEOMETRY object)	the JSON object for that geometry
	SDO_UTIL.FROM_GEOJSON	a geometry in GEOJSON format (as a VARCHAR2 or as a CLOB or as a JSON), crs parameter (as a VARCHAR2, now by default set to NULL, reserved for future use) and a SRID (as a VARCHAR2, by default set to 4326)	the geometry value for that GeoJSON representation (as an SDO_GEOMETRY object)
	SDO_UTIL.FROM_JSON	a geometry in JSON format (as a JSON object or as a CLOB), crs parameter (as a VARCHAR2, now by default set to NULL, reserved for future use) and a SRID (as a VARCHAR2, now by default set to -1, reserved for future use)	the geometry value for that JSON representation (as an SDO_GEOMETRY object)
KML	SDO_UTIL. TO_KMLGEOMETRY	a geometry value (as an SDO_GEOMETRY)	the geometry value in KML language (as a CLOB)
	SDO_UTIL. FROM_KMLGEOMETRY	a geometry in KML language (as a VARCHAR2 or a CLOB)	the geometry value for that KML representation (as an SDO_GEOMETRY object)

TABLE 7. Functions related to GeoHash format from SDO_CS package in Oracle XE 21c (based on [49])

Function Name	Input Arguments	Output
SDO_CS.TO_GEOHASH	an element of SDO_GEOMETRY type and geohash length (as a NUMBER)	the geohash value for that geometry (as a VARCHAR2)
SDO_CS. FROM_GEOHASH	a geohash value (as a VARCHAR2) and a SRID (as a NUMBER)	a geometry value of SDO_GEOMETRY type representing a specified geohash

TABLE 8. Functions related to WKT and EWKT formats in PostGIS 3.4 (based on [55])

Format	Function Name	Input Arguments	Output
WKT	ST_AsText	a geometry value of a geometry or geography type (or a geometry value and additional parameter used to reduce the maximum number of decimal digits after floating point used in output)	the WKT representation of the given geometry (as a text)
	ST_WKTToSQL (this is an alias name for ST_GeomFromText that takes no SRID)	a WKT representation of a geometry value (as a text)	the geometry value for that WKT representation (as element of a geometry type)
	ST_GeomFromText and ST_GeometryFromText	a WKT representation of a geometry value (as a text) (then SRID is default equal to 0) or a WKT representation of a geometry value and a SRID (as an integer)	the geometry value for that WKT representation (as element of a geometry type)
	ST_PointFromText, ST_LineFromText, ST_PolygonFromText, ST_GeomCollFromText, ST_MPointFromText, ST_MLineFromText, ST_MPolyFromText	a WKT representation of geometry value of an appropriate type (as a text) (then SRID is default equal to 0) or a WKT representation of a geometry value and a SRID (as an integer)	the geometry of an appropriate type for that WKT representation (i.e. POINT, LINESTRING, POLYGON, GEOMETRYCOLLECTION, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, respectively) as an element of geometry type, or null if the WKT is not describing the geometry from an appropriate class
	ST_BdPolyFromText	a WKT representation of a MultiLineString for an arbitrary collection of closed linestrings (as a text) and a SRID (as an integer)	the Polygon for that WKT representation (as an element of a geometry type)
	ST_BdMPolyFromText	a WKT representation of a MultiLineString for an arbitrary collection of closed linestrings (as a text) and a SRID (as an integer)	the MultiPolygon for that WKT representation (as an element of a geometry type)
EWKT	ST_AsEWKT	a geometry value of a geometry or geography type and some optional parameter with maximum number of decimal digits in output ordinates	the EWKT representation of the given geometry (that is a WKT representation with SRID metadata) (as a text)
	ST_GeomFromEWKT	a EWKT representation of a geometry value (as a text)	the geometry value for that EWKT representation (as element of a geometry type)
WKT or EWKT	ST_GeogFromText and ST_GeographyFromText	a EWKT or a WKT representation of a geometry value (as a text)	the element of a geography type (with SRID 4326 if it is unspecified) for that representation

TABLE 9. Functions related to WKB, EWKB, HEXEWKB and TWKB formats in PostGIS 3.4 (based on [55])

Format	Function Name	Input Arguments	Output
WKB	ST_AsBinary	a geometry value of a geometry or geography type (or a geometry object and some encoding parameter)	the WKB representation of the given geometry (as a bytea)
	ST_WKBToSQL (this is an alias name for ST_GeomFromWKB that takes no SRID)	a WKB representation of a geometry value (as a bytea)	the geometry value for that WKB representation (as element of a geometry type)
	ST_GeomFromWKB	a WKB representation of a geometry value (as bytea) (then SRID is default equal to 0) or a WKB representation of a geometry value and a SRID (as an integer)	the geometry value for that WKB representation (as element of a geometry type)
	ST_PointFromWKB, ST_LineFromWKB or ST_LinestringFromWKB	a WKB representation of a geometry value (as bytea) (then SRID is default equal to 0) or a WKB representation of geometry value and a SRID (as an integer)	the geometry of an appropriate type (POINT, LINESTRING) for that WKB representation as an element of a geometry type, or null if the WKB is not describing the geometry from an appropriate class
EWKB	ST_AsEWKB	a geometry value of a geometry type (or a geometry value and some encoding parameter)	the EWKB representation of the given geometry (that is a WKB representation with SRID metadata) (as a bytea)
	ST_GeomFromEWKB	a EWKB representation of a geometry value (as bytea)	the geometry value for that EWKB representation (as element of a geometry type)
WKB or EWKB	ST_GeogFromWKB	a EWKB or a WKB representation of a geometry value (as a bytea)	the geometry value for that representation (as element of a geography type)
HEXEWKB	ST_AsHEXEWKB	a geometry value of a geometry type (or a geometry value and some encoding parameter)	the HEXEWKB representation of the given geometry (that is in the hexadecimal numeral system) (as a text)
TWKB	ST_AsTWKB	a geometry value of a geometry type (or an array with a geometry collection) and some additional parameters	the TWKB representation of the given geometry (as a bytea)
	ST_GeomFromTWKB	a TWKB representation of a geometry value (as a bytea)	the geometry value for that TWKB representation (as element of a geometry type)

TABLE 10. Functions related to GML, GeoJSON, KML and GeoHash formats in PostGIS 3.4 (based on [55])

Format	Function Name	Input Arguments	Output
GML	ST_AsGML	a geometry value of a geometry or geography type and some additional parameters (one of the parameters concerns version of GML, may be either 2 or 3, 2 is a default value)	the GML representation of the given geometry value (as a text)
	ST_GMLToSQL and ST_GeomFromGML	a GML representation of a geometry value (as a text) or a GML representation of a geometry value and SRID as (an integer)	the geometry value for that GML representation (as a geometry)
GeoJSON	ST_AsGeoJSON	a geometry value of a geometry or geography type or record feature and the name of geometry column (and some optional parameters)	the GeoJSON representation for that geometry (a GeoJSON geometry object, or the row as a GeoJSON feature object, both written as a text)
	ST_GeomFromGeoJSON	a GeoJSON representation of a geometry value (as a text, json or jsonb)	the geometry value for that GeoJSON representation (as a geometry)
KML	ST_AsKML	a geometry value of a geometry or geography type and some optional parameters	KML representation of the given geometry value (as a text)
	ST_GeomFromKML	a KML representation of a geometry value (as a text)	the geometry value for that KML representation (as a geometry)
GeoHash	ST_GeoHash	a geometry value of a geometry type and some optional parameter	the GeoHash representation of the geometry (as a text)
	ST_GeomFromGeoHash	a GeoHash value (as a text) and some optional parameter	the geometry value for that GeoHash (as a geometry)
	ST_Box2dFromGeoHash	a GeoHash value (as a text) and some optional parameter	the geometry value for that GeoHash (as a box2d)
	ST_PointFromGeoHash	a GeoHash value (as a text) and some optional parameter (telling how many characters from the GeoHash is used to create the point)	the point that represents the center point of the geometry from a given GeoHash value

TABLE 11. The availability of functions supporting a given format in MySQL 8.0, Oracle XE 21c and PostGIS 3.4/PostgreSQL 14

Format	MySQL	Oracle XE	Postgis/PostgreSQL
WKT	✓	✓	✓
EWKT			✓
WKB	✓	✓	✓
EWKB			✓
TWKB			✓
HEXEWKB			✓
GML		✓	✓
GeoJSON	✓	✓	✓
GeoHash	✓	✓(with some problems)	✓
KML		✓	✓