Computers & Geosciences 155 (2021) 104840

Contents lists available at ScienceDirect

Computers and Geosciences

journal homepage: www.elsevier.com/locate/cageo

Parallel algorithm for improving the performance of spatial queries in SQL: The use cases of SQLite/SpatiaLite and PostgreSQL/PostGIS databases

Mateusz Ilba [1]

Department of Social and Economic Geography, Cracow University of Economics, Cracow, Poland

ARTICLE INFO

Keywords:
SpatiaLite
PostGIS
Algorithms
Spatial data processing
Parallel and high performance computing

ABSTRACT

This paper proposes an open-source algorithm that performs parallel processing of spatial queries, during which an initial selection of objects to be subjected to spatial relationship tests is done using a spatial index. These data are then further subdivided by the use of the OFFSET and LIMIT clauses into still smaller subgroups, to which spatial relationship tests utilizing complex calculations are assigned, thereby creating multiple processes running in parallel. This algorithm was tested using data from the SQLite/SpatiaLite and PostgreSQL/PostGIS database. In processing spatial relationship queries involving six threads, the algorithm yielded a 3.6X maximum speed-up increase in performance compared to single-thread processing on SQLite/SpatiaLite database and 5.1X maximum speed-up on PostgreSQL/PostGIS database. In single-layer analyses (e.g., area calculation, buffer generation), a 5X speed-up time in query processing was observed.

1. Introduction

For many years, spatial object databases have been used to store, manage, and analyze spatial data. Employing these is more efficient than using traditional file resources (e.g., data collected in shapefile or other files). In addition to dataset update operations, performing spatial analyses using such databases is also more effective (Agarwal and Rajan, 2016). Contemporary spatial databases are more than just databases: Thanks to the existence of query languages (e.g., SQL), these also constitute a full-fledged application for spatial analysis.

To significantly increase the efficiency of spatial analyses performed with such databases, spatial object indexing systems based on mutual object relationships were introduced. Despite the use of object indexing systems, analysis of large spatial datasets is a demanding task for a single computer, and so special parallel-processing platforms have been created to handle large datasets (e.g., the MapReduce framework (Dean and Ghemawat, 2008) in Hadoop (Shvachko et al., 2010)). Processing and analysis of data are based on the distribution of calculations to many individual computers (i.e., computer clusters), thereby enabling analyses based even on trillions of objects to be completed in an acceptable time. Distributed platforms also enable the analysis of spatial data (ESRI, 2019; Giannousis et al., 2018; Hagedorn and Räth, 2017; He et al., 2019; Karthi and Prabu, 2018; Priya and Kalpana, 2018; You et al., 2015). For example, distributed computing systems are used to manage and analyze large climate spatial datasets (Hu et al., 2018) and perform seismic signal analyses (Addair et al., 2014). However, the use of such systems is not always economically viable, and smaller databases can be successfully processed within one computer.

In contrast to distributed solutions, easy-to-use databases of spatial objects, such as SQLite/SpatiaLite and PostgreSQL/PostGIS (PostGIS 3 supports multicore sequence scans, aggregates, and joins [Ramsey, 2019]) make it possible to execute common spatial queries in a single thread. Consequently, during a single query, one computer thread (i.e., computing core) is used to process the query. Nowadays, every new processor has from several to as many as ten or more cores, which are typically not used when executing advanced SQL queries. The current development of hardware aimed at creating the multicore processor has thus motivated the author of this paper to determine how to use all of a single computer's resources to decrease the processing time of SQL queries used for GIS analyses.

The main objective of the research presented in this paper was to test the possibility of multicore processing of single SQL queries related to spatial analyses in the SQLite/SpatiaLite and PostgreSQL/PostGIS database. Therefore, an algorithm was written in the Python language that calls a parallel SQL query within a single database. The performance of the resulting solution was assessed by distinguishing between different types of objects involved in the analysis, their number, and the number of threads used. In addition, also evaluated was whether the

E-mail address: ilbam@uek.krakow.pl.

[1] Link to the code: https://github.com/MateuszIlba/SpatiaLite-parallel-processing , https://github.com/MateuszIlba/PostGIS-parallel-processing.

https://doi.org/10.1016/j.cageo.2021.104840
Received 4 March 2020; Received in revised form 25 May 2021; Accepted 29 May 2021
Available online 11 June 2021

speed of the data storage medium on which the database was stored had an impact on the speed of parallel processing. The following characteristics of the SQLite/SpatiaLite and PostgreSQL/PostGIS database motivated the author to employ it in evaluating the proposed algorithm's performance:

- Easy to use, possibility to use standard SQL queries and functions;
- Distributed computer network not required;
- Dividing the data among many computers not an issue (e.g., In distributed systems, it is difficult to divide data among many computers to check the topological relationships of the spatial objects located in an entire area);
- Possibility of integration with the environment for visualization of results of spatial queries in an open QGIS application.

The remainder of the paper is organized as follows. Section 2 briefly summarizes previous research related to the proposed algorithm, and Section 3 discusses the methods used in developing the algorithm. Section 4 describes the experiments conducted to evaluate the performance of our proposed approach, and Section 5 then discusses the results of these experiments. Finally, Section 6 concludes our presentation and suggests avenues for future work.

2. Related work

Studies using multithreading in various applications and involving spatial databases have been conducted for a variety of purposes, ranging from data storage itself (i.e., increasing the efficiency of data import/export) to data analysis. In addition, some published research concerns the use of multithread processing based on Graphics Processing Unit (GPU) solutions. However, the majority of solutions are based on distributed processing using the MapReduce framework, e.g., Hadoop and similar systems.

Research on multithread spatial joins in databases can be found in many published studies (e.g., Brinkhoff et al., 1996; Hoel and Samet, 1994; Zhou et al., 1998), which describe the benefits that can be derived from implementing parallel algorithms. For instance, Papadopoulos and Manolopoulos (2003) describe the implementation of parallel algorithms for bulk-loading index structures and spatial data in shared-nothing architectures, demonstrating the benefits of using multicore processors for this purpose. On the other hand, Kim et al. (2013) present the R -tree traversal algorithm, which enables searching for objects based on location, and note that an increase in the number of Central Processing Unit (CPU) threads correlates with increase performance in query execution; however, a query run on 24 CPU cores yielded only a 3-fold performance improvement over one run on two CPU cores. The doctoral dissertation of Dai (2009), which also presents methods of optimizing spatial indices, discusses the problem of parallel queries based on R-tree indexing and linear spatial indexing.

Due to the much greater capacity of parallel processing on the GPU, interest in its use in database processing has also increased. Shehab et al. (2017) describes how to split an SQL query into smaller parts in relational databases using the example of Microsoft SQL Server, where the use of the GPU enables a 39-fold increase in performance. Zhang et al. (2014) present a solution intended to speed up the analysis of data aggregation concerning taxi pickup location points and streets in a linear form. On the other hand, is an example of an Nvidia CUDA computing platform, Simion et al. (2012) show that query processing speed can be increased in individual cases from 62 to as much as 318 times. Parallel computations of the GPU, such as the PostgreSQL database extension, can also support three-dimensional object databases (Real and Silva, 2018). While the use of GPU-based solutions yields the greatest benefit in parallel processing, the use of all available CPU threads also yields huge performance benefits to a database system.

Ray et al. (2013) presented the parallel spatial data analysis in PostgreSQL/PostGIS using Amazon EC2 instances. They observed 63.4X speed-up in the best case, using 64-core (on 16 4-core EC2 nodes), but in the worst case, they observed only a 7.3x increase in speed. They used spatial declustering of spatial data for all nodes. An infrastructure called Niharika creates balanced spatial partitions of data and is very important for the optimal load on all the nodes/cores.

SQLite is a relational database management system also used with parallel components. Limkar and Jha (2019) describe the framework for parallel computing of spatial data (i.e., IoT generated, encrypted). They propose using parallel construction of index R-trees, R + -tree and R*-tree method indexing. Alomari et al. (2019) present the SQLite-XTS parallel database encryption system, describing parallel data encryption on mobile devices using the SQLite database system. Kai (2016) discusses a shared memory-based lock manager, which effectively allows writing concurrent threads, and describe how the use of a multicore allowed pick up of transactions per minute to increase from 47% to 110% (depending on CPU frequency). Cremer et al. (2017) present an upgrade to SQLite named CuDB with hybrid CPU/GPU processing, which, in substring searches on unindexed tables, provided increased speeds of 400X over those obtained with SQLite.

Based on the results provided in the examples above, SQLite and PostgreSQL is suitable for multithreaded computation.

3. Methods

In this section, how to address multi-threaded processing on standard SQL queries was presented. Section 3.1 describes how data was split, based on a spatial indexing system, an SQL LIMIT, and OFFSET clauses. The next section, 3.2, presented how a query based on spatial indexing was performed for both spatial database systems (SQLite/SpatiaLite and PostgreSQL/PostGIS). This section also described the main parts of the Python algorithm that was used.

3.1. Splitting input data

In formulating the proposed algorithm, the main problem the author faced was splitting the query input data into subgroups for processing by separate processor cores. This process is, in general, not trivial and must be carried out efficiently. An example of data splitting is grouping (Dobos et al., 2013; Zhang and You, 2012), which can be efficient but, in principle, is limited to point data. Data splitting using tiling is a popular way to operate large raster datasets in parallel (Aji et al., 2012; Vinhas et al., 2003).

Another means of splitting data for the distribution of parallel processing is used in distributed processing based on range, e.g., using the MapReduce implementation on the Hadoop platform (Aji et al., 2015; Romero et al., 2015). However, this method requires reserving a part of the database on a hard drive or storing it in volatile memory. In simple spatial database systems that are not adapted to parallel processing, this constraint is a significant impediment to database updates. A spatial coding-based approach is another way of partitioning big spatial data on the Hadoop platform (Yao et al., 2017) and, compared to random sampling, can improve query performance.

The strategy chosen by the author is based on the equal division of the number of records pre-selected for analysis by the spatial index.

Publicly available database systems contain different types of such indexing systems. For example, the IBM DB2 system (Adler, 2001) employs two object indexing methods: one using spatial grid indexes for flat data and Geodetic Voronoi indexes for geodetic data in a spherical projection. The PostgreSQL/PostGIS and Oracle Spatial systems use a hierarchical R-Tree spatial index (Guttman, 1984) system based on a bounding box. The SQLite/SpatiaLite system uses the VirtualSpatialIndex (SpatiaLite, 2017a), which is a development of the powerful R*-tree system (Beckmann et al., 1990). Proprietary indexing systems have also been created, such as ones based on and enjoying the benefits of saving databases on SSD drives (for example, the efficient framework system, called eFIND (Carniel et al., 2019) and the flash-optimized unbalanced R-tree index (Jin et al., 2015)) and used to process efficient spatial queries in wireless communication environments (Park, 2014), analyze BIM data (Solihin et al., 2017), or manage LiDAR data in Oracle Spatial (Schön et al., 2013). Each indexing system differs about creation time, the number of references to the original data, and speed of operation (Roumelis et al., 2017).

A spatial index based on the XY location of objects is the means most often employed to speed up queries. The R*-tree index, which represents an improvement of the R-tree index (Guttman, 1984), was developed by N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger (Beckmann et al., 1990). A spatial index based on the R*-tree is constructed of hierarchically arranged bounding boxes (minimum and maximum XY) of spatial objects (Fig. 1). Through a tree constructed of bounding boxes, it is possible to quickly find pairs of objects having probable relationships (e.g., intersects). R*-tree is attractive because it is efficient, it supports spatial data and points at the same time, and its creation time is slightly higher than those of other R-tree variants (Liu et al., 2020).

Parallel processing is performed by invoking the same SQL query on different parts of a dataset. In the proposed algorithm, these parts are split using the SQL LIMIT and OFFSET clauses (SQLite, 2020), which enable the selection of a specific part of a dataset without necessitating disk-writing operations. Additionally, records to be analyzed are preliminarily limited by the spatial indexing implemented in the SQLite/SpatiaLite and PostgreSQL/PostGIS database.

Because of the parallel execution of SQL queries, the proposed method returns the result of the SQL query in separate datasets in the form of variables that can be combined into one whole dataset and saved to a disk in a separate file or a database participating in the analysis being performed.

3.2. *Multicore SQL spatial query algorithm*

The algorithm proposed by the author was written in the Python programming language (Python version 2.7.16). To run the algorithm on a computer with SQLite/SpatiaLite database, installation of the *sqlite3* library extension (SpatiaLite 4.4.0) called *mod_spatialite* (SpatiaLite, 2017b) is necessary. On a computer with PostgreSQL (9.6) PostGIS (2.3.0) database, installation *psycopg2* library extension (Di Gregorio and Varrazzo, 2020) is require.

The first part of the algorithm determines how many objects will be processed by a single thread of the analysis, a number that can be determined by counting all objects and dividing the result by the expected number of threads to be used for the analysis.

For analyses based on a single layer of spatial objects (e.g., buffering, area calculation, length, etc.), counting the number of rows of the layer using the *COUNT(*)* function is sufficient. In an analysis based on the topology of two layers (e.g., intersects(), touches(), within(), crosses(), etc.), the number of rows is estimated by way of a preliminary selection based only on a spatial index. Examples of using a spatial index in SQLite/SpatiaLite are as follows:

```
select *
from firstlayer
inner join secondlayer on
      secondlayer.ROWID IN (
      SELECT ROWID
      FROM SpatialIndex
      WHERE f_table_name = secondlayer '
      AND search_frame = firstlayer.Geometry)
```

Examples of using a spatial index in PostgreSQL/PostGIS are as follows:

```
SELECT *
FROM firstlayer
JOIN secondlayer
ON firstlayer.geom && secondlayer.geom
```

The selection creates a new set of object pairs that can be subjected to a logical test by the spatial function, and the object pairs can be equally distributed among all processes by defining the limit of the number of objects to be allocated to a single process as follows:

$$\text{LIMIT} = \text{int}(\text{rowcount}/\text{process}) + 1$$

Due to the occurrence of non-integer values resulting from the division and the required certainty of use of all object pairs from the set, the value 1 is added to the calculated number.

The value of the OFFSET for the individual threads of the analysis is calculated based on the LIMIT value; the first value for the first process is 0, for the second process the first value is increased by the LIMIT value, and, for subsequent n-processes, the first value is increased by (n - 1) * LIMIT value. An illustrative example of the operation of the OFFSET and LIMIT clauses for three parallel processes in a dataset comprised of nine elements is shown in Fig. 2.

The execution of parallel processing of an SQL query on parts of a dataset prepared for analysis was carried out by the Python *multiprocessing* package (Python Documentation, 2019). The multiprocessing package allows remote and local concurrency and uses subprocesses instead of threads. The multiprocessing module allows multicore processors (and also multiple processors) to be fully leveraged on a given computer. The code for using the package takes the following form:

The result is stored in the results variable in the form of a list of objects or values (depending on the SQL query type). Fig. 3 shows the

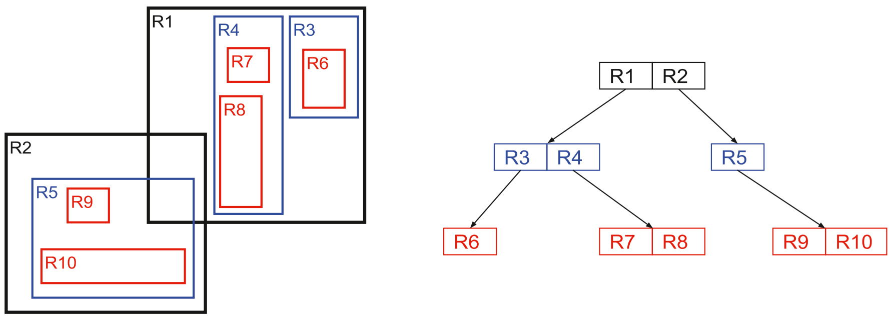

Fig. 1. A simple example of an R-tree for minimum bounding rectangles.

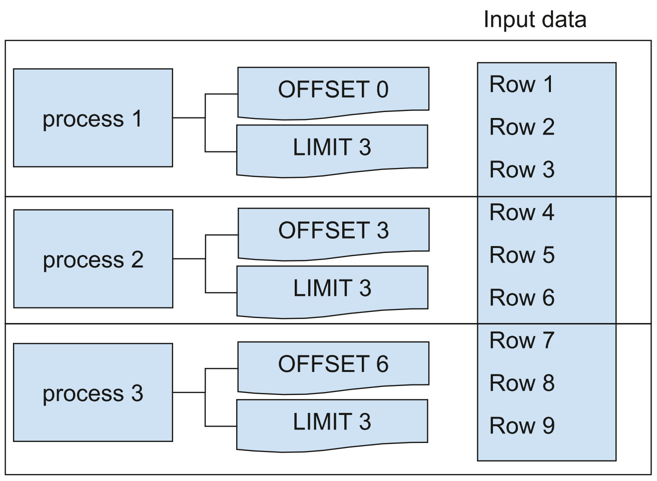

Fig. 2. Diagram showing the operation of the LIMIT and OFFSET clauses in the proposed data splitting solution for analysis on 3 threads.

flow diagram of the developed algorithm.

The entire program code is limited to the main functions: row_count () – count of the number of all object pairs to be analyzed based on their topology; gen_list_offset() - the number of line offsets to split the analysis into threads; and main_query() - the function of the main SQL query. The program codes are available under an open license on the GitHub platform (https://github.com/MateuszIlba/SpatiaLite-parallel-processing, https://github.com/MateuszIlba/PostGIS-parallel-processing), and the complete codes are given in Appendix A and C.

4. Experimental evaluation

4.1. Experimental data

To test the algorithm, databases were created from publicly available OpenStreetMap data for Poland (Ilba, 2020). Processing performance was tested using four databases of different sizes, the structures of which are shown in Table 1. The tests were conducted in a Windows system on a computer equipped with an i5-8400 processor (6 cores, 2.8 GHz, 3.8 GHz boost) with 16 GB of RAM. The databases were stored on a GoodRAM IRDM Pro 256 GB SSD drive, and the data were obtained in the form of *.shp files from Geofabrik (2018).

Databases contain data from the area:

- The db_small - area of the city of Krakow;
- The db_medium - entire Malopolskie Province;
- The db_large - Malopolskie, Podkarpackie and Dolnoslaskie Provinces;
- The db_v_large - covers the entire Poland country.

4.2. Performance testing method

The tests were selected based on the tests performed in the Jackpine benchmark, in which spatial object databases were compared (Ray et al., 2011). Spatial queries based on the topological relations of objects were selected—intersects(), touches(), within(), and crosses() between different layer combinations: lines – lines (layer1 = 'roads_lines', layer2 = 'waterways_lines'), polygons – polygons (layer1 = 'landuse_polygon', layer2 = 'pois_polygon'), polygons – point (layer1 = 'landuse_polygon', layer2 = 'buildings_points'), lines – polygons (layer1 = 'roads_lines',

```
def main():

        pool = mp. Pool( processes= mp. cpu_count() )

        results = [ pool.map(SQL query function, (list of offset)) ]
```

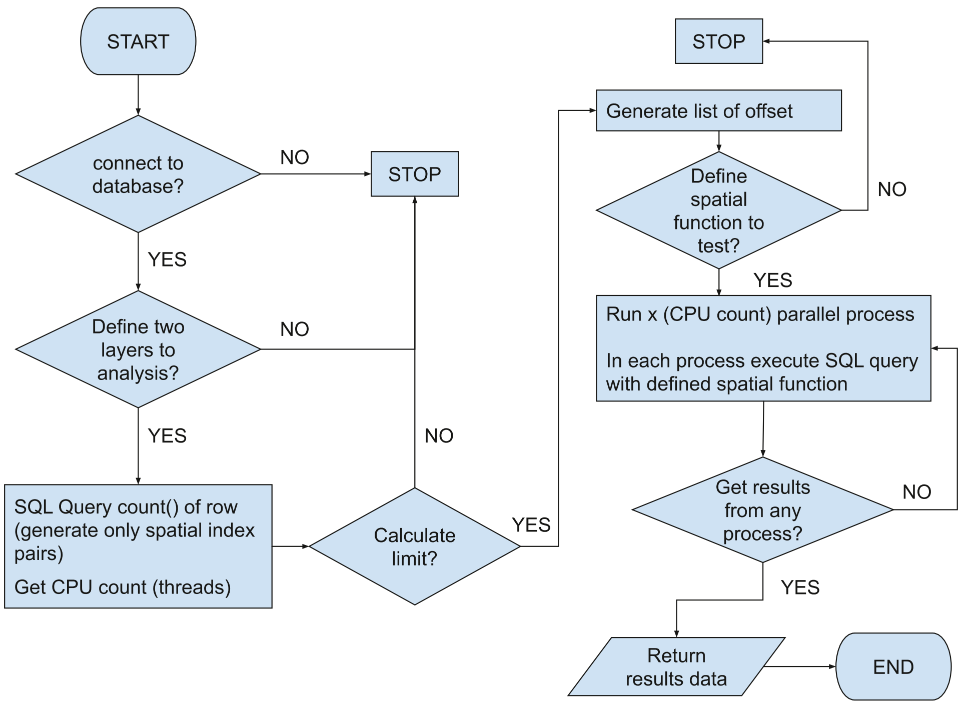
START
connect to database?
NO
STOP
YES
Define two layers to analysis?
NO
YES
NO
SQL Query count() of row (generate only spatial index pairs)
Get CPU count (threads)
Calculate limit?
YES
STOP
Generate list of offset
Define spatial function to test?
NO
YES
Run x (CPU count) parallel process
In each process execute SQL query with defined spatial function
Get results from any process?
NO
YES
Return results data
END

Fig. 3. Flow diagram of the operation of the program for parallel processing of SQL queries using the SQLite/SpatiaLite and PostgreSQL/PostGIS database system.

Table 1
Description of the databases used to test the performance of the algorithm.

Name of database	Layer *buildings_points* count:	Layer *landuse_polygon* count:	Layer *pois_polygon* count:	Layer *roads_lines* count:	Layer *waterways_lines* count:
db_small (47 MB[a])	113 870	15 513	3 396	63 943	1 083
db_medium (307 MB[a])	1 049 360	94 005	12 997	265 336	16 075
db_large (962 MB[a])	3 154 383	345 556	39 321	780 111	73 596
db_v_large (3550 MB[a])	11 157 533	1 627 622	164 290	3 122 340	157 711
Type of geometry	point	polygon	polygon	multilinestring	multilinestring
Source	osm_buildings	osm_landuse	osm_pois	osm_roads	osm_waterways

[a] The size of database on disk.

layer2 = 'landuse_polygon'), and lines – points (layer1 = 'roads_lines', layer2 = 'buildings_points').

The query execution time was measured from the start of the query until the result was obtained in the form of the number of objects meeting a specific spatial relationship (i.e., Count). The tests did not take into account the time spent possibly recording the result of the analysis in the database since parallel recording in SQLite databases is not possible. Instead, the result of the analysis must be recorded in a single thread. Speed-up was calculated by the formula:

$$Speed\text{-}up = \frac{compl.\ time\ of\ multi\ core\ processing\ [s]}{time\ of\ single\ core\ processing\ [s]}$$

It follows that values below 1 signify a slowdown compared to the traditional SQL query.

4.3. Performance results

Tables 2 and 3 show all the times of the analyses based on single-core and multi-core query processing, and the time values apply to the entire query. In the case of multi-threaded processing, queries based on a spatial index (to determine the limit and offset) were also included.

Fig. 4 shows the performance test results in the form of values indicating how much performance increased or decreased with all processor cores (six threads) used in contrast to traditional single-thread processing. The values used for the graphs were obtained by dividing the single-thread processing time by the multi-thread processing time.

The researcher also checked the impact of the number of threads on the increase in the speed of analysis. For this purpose, the lines-polygons touches analysis was selected, and the number of threads used during that analysis was changed (Fig. 5).

Table 4 shows the selected time cost (in seconds) for different parts of the algorithm running on the SQLite/SpatiaLite database. Appendix B and D show all results of all tests. The first part of the algorithm is a function that estimates the data to be subjected to the LIMIT and OFFSET clauses and that is based on applying the COUNT() function to obtain all pairs in spatial joins (based on the spatial index only). The algorithm's second part realizes the number of the selected SQL queries parallel by all cores (the data for specified topological tests were split by using the LIMIT and OFFSET clauses).

A test consisting of an analysis based on one layer only, of generation of a buffer around the layer, and of calculation of its surface was also performed. For each of the five layers from the largest database, the performance of a 6-thread analysis increased by up to five times that of a single-thread analysis (Fig. 6). The effect of database size was also analyzed and is presented in Fig. 7.

The last element that the researcher checked was the impact of the data storage device (i.e., the location where the SQLite/SpatiaLite database was saved) on the multithreaded processing performance. The largest database was saved to an HDD (512 GB, 5400 rpm, up to 100 MB/s read/write), an SSD (256 GB, SATA 3, MLC, up to 500 MB/s read/

Table 2
Analysis times in seconds [s] for the cases under consideration in SQLite/SpatiaLite. The pairs of values in which a multi-core query is faster are marked in green. Time of multi-core processing is a final time cost (with pre-selecting pairs by spatial index). All cases can be seen in Appendix B.

Name of	Type of	L - L[a]		PO - PO[b]		PO - PK[c]		L - PO		L - PK	
database	analysis	single*	multi**	single	multi	single	multi	single	multi	single	multi
db_small	crosses	3.0	3.3	2.6	2.2	1.2	1.7	7.2	8.2	14.8	11.9
	intersects	0.7	2.7	1.9	2.1	0.7	1.6	6.5	8.0	8.5	10.7
	touches	3.0	3.3	2.6	2.3	1.2	1.7	7.2	8.1	14.9	11.9
	within	0.6	2.7	1.7	2.0	0.5	1.6	5.6	7.9	7.4	10.6
db_medium	crosses	106	49	16	15	532	244	235	78	125	97
	intersects	61	39	10	12	58	69	144	58	48	84
	touches	106	48	16	15	524	236	235	77	125	96
	within	94	47	8	11	40	64	172	64	40	81
db_large	crosses	379	163	61	54	1 619	642	813	324	392	220
	intersects	202	125	38	45	180	116	444	214	146	170
	touches	396	161	59	53	1616	646	814	330	397	217
	within	343	153	32	43	124	100	604	284	400	219
db_v_large	crosses	1 316	586	428	273	14 593	4 536	7 242	2 107	1 462	830
	intersects	694	466	197	226	1 399	782	3 358	1 345	595	723
	touches	1 379	587	426	269	14 574	4 538	7 428	2 076	1 369	821
	within	1 183	565	169	229	1 042	716	6 064	1 819	481	664

[a] Lines, [b] Polygons, [c] Points, * single-core processing, ** multi-core processing

Table 3
Analysis times in seconds [s] for the cases under consideration in PostGIS. The pairs of values in which a multi-core query is faster are marked in green. Time of multi-core processing is a final time cost (with pre-selecting pairs by spatial index). All cases can be seen in Appendix D.

Name of	Type of	L - L[a]		PO - PO[b]		PO - PK[c]		L - PO		L - PK	
database	analysis	single[*]	multi[**]	single	multi	single	multi	single	multi	single	multi
db_small	crosses	5.0	1.5	2.9	1.3	53.7	18.4	39.0	11.0	17.0	7.6
	intersects	0.5	0.5	1.2	0.9	1.1	1.4	21.6	4.9	3.6	4.9
	touches	5.0	1.5	2.9	1.3	55.8	18.5	38.9	11.0	17.0	7.4
	within	3.7	1.3	1.0	0.9	0.6	1.2	26.4	5.8	1.6	4.5
db_medium	crosses	126	37	17	5	1084	433	382	177	197	111
	intersects	6	4	5	3	16	26	30	15	43	82
	touches	127	37	17	7	1079	361	383	123	197	111
	within	103	32	2	2	10	25	13	11	28	79
db_large	crosses	462	117	67	24	3 543	1 049	1 379	407	787	271
	intersects	26	17	18	12	300	68	113	58	700	219
	touches	465	116	67	23	3 530	1 054	1 340	354	752	266
	within	379	105	9	5	209	75	44	44	187	155
db_v_large	crosses	1 605	364	600	203	32 207	6 284	12 883	3 763	2 883	962
	intersects	71	45	96	124	4 029	925	687	418	2 603	836
	touches	1 595	371	593	211	32 207	6 313	12 453	3 706	2 886	961
	within	1 313	310	49	64	1 979	456	302	318	891	587

[a] Lines, [b] Polygons, [c] Points, [*] single-core processing, [**] multi-core processing

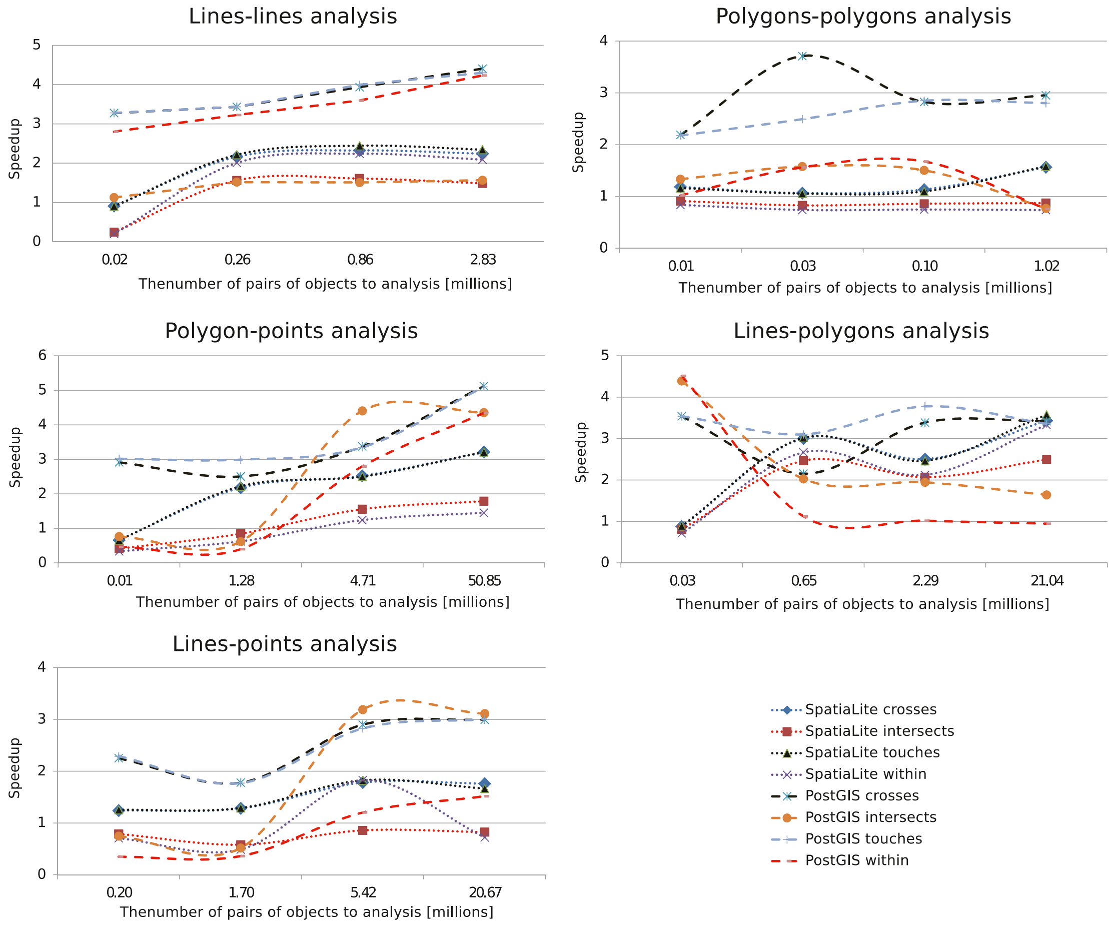

Fig. 4. Results of the performance of the analysis of intersects, touches, within and crosses queries performed with 6 threads (the value in the graphs shows how much faster (speed-up) or slower (value below 1) the query was executed compared to single-thread processing).

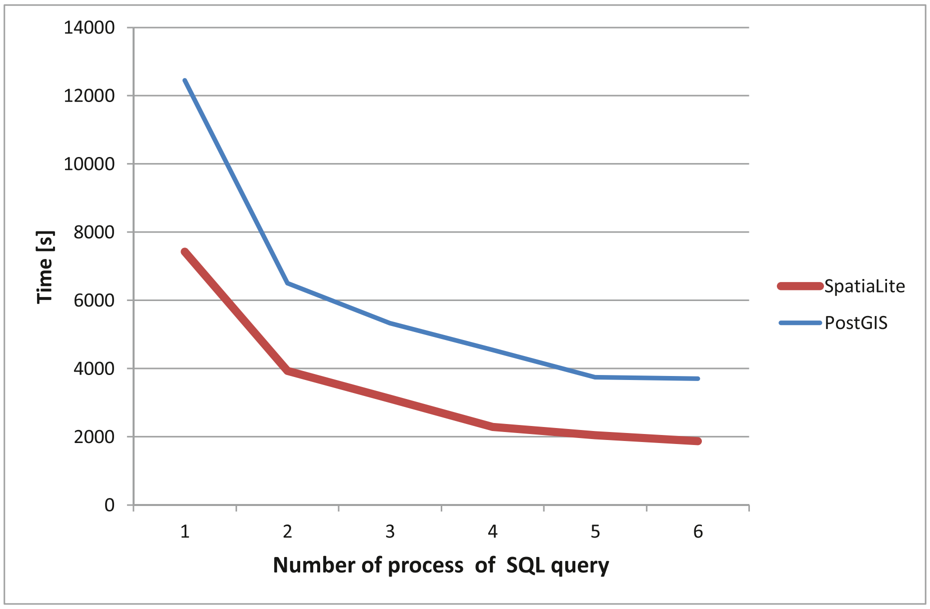

Fig. 5. Time of analysis of lines-polygons touches concerning the number of processes used.

Table 4
Selected time [s] costs while the SQL query is processed (touches() function) in SQLite/SpatiaLite. Time of final multi-core query processing consists of time for estimate count of row (based on the spatial index) and time of multi-core processing (using LIMIT and OFFSET).

Name of database	Layers	Single-core query processing	Count of row based on spatial index	Multi-core query processing	Final multi core time cost
db_small	L - L[a]	**3.0**	2.3	1.0	**3.3**
	PO - PO[b]	**2.6**	0.6	1.7	**2.3**
	PO - PK[c]	**1.2**	1.0	0.7	**1.7**
	L - PO	**7.2**	2.8	5.3	**8.1**
	L - PK	**14.9**	3.2	8.7	**11.9**
db_V_large	L - L[a]	**1 379**	126	461	**587**
	PO - PO[b]	**426**	73	196	**269**
	PO - PK[c]	**14 574**	206	4 332	**4 538**
	L - PO	**7 428**	203	1 873	**2 076**
	L - PK	**1 369**	207	614	**821**

[a] Lines.

[b] Polygons.

[c] Points.

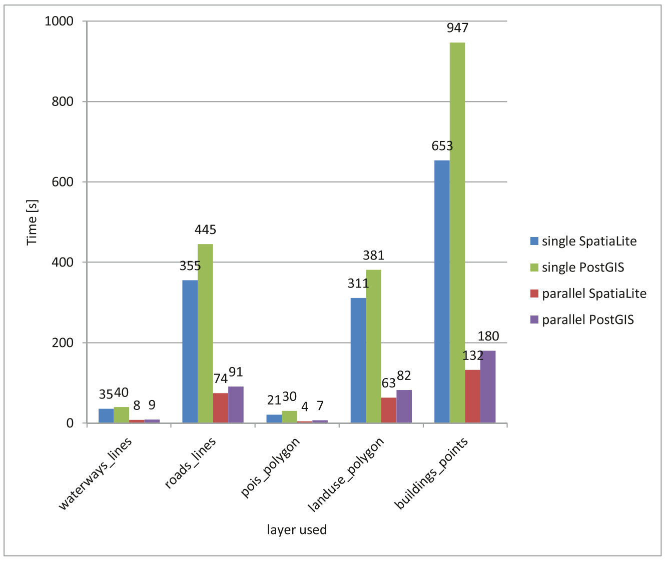

Fig. 6. The time of calculation of the area of a buffer generated around all objects of a layer (on db_v_large).

write), and a RAM-disk (4 GB, up to 10 000 MB/s read/write). Speed of query processing was tested on db_v_large, and an intersect query of two layers was used, i.e., landuse_polygon and buildings_points. The differences in processing speed were negligible and were as follows: HDD – 769 [s], SSD – 775 [s], and RAM-disk – 769 [s].

5. Discussion

The multithread-processing performance yielded a maximum 3.6X speed-up in the analysis time in processing a touches() query on SQLite/SpatiaLite for lines with polygon object pairs and 5.1X speed-up in the analysis time in processing crosses() and touches() query on PostgreSQL/PostGIS for polygons with points object pairs. Tarmur and Özturan (2019) present work about classification points in regions (spatial joins) using Apache Spark. Using spatial functions and cloud computing, they obtained a 1.6- to 4.5-fold increase in speed compared to that of Microsoft SQL Server. Performance gains in tasks that required spatial topology were close to those shown in the researcher's results in parallel SQLite/SpatiaLite and PostgreSQL/PostGIS spatial joins (e.g., point and polygon spatial intersections).

The impact of the number of threads, ranging from 1 to 6, on the acceleration of the analysis was also tested (Fig. 5). A clear slowdown in performance occurred as the number of threads used for the analysis increased. Moreover, tests on larger numbers of threads (i.e., 8, 16, and 24) in the STARK, Geospark, and Spatial-Spark systems has shown small performance increases (Giannousis et al., 2018). The explanation for this nonlinear increase in performance was hypothesized to be the uneven distribution of computing tasks between individual cores. The groups of object pairs created for analyses of spatial relationships are typically not

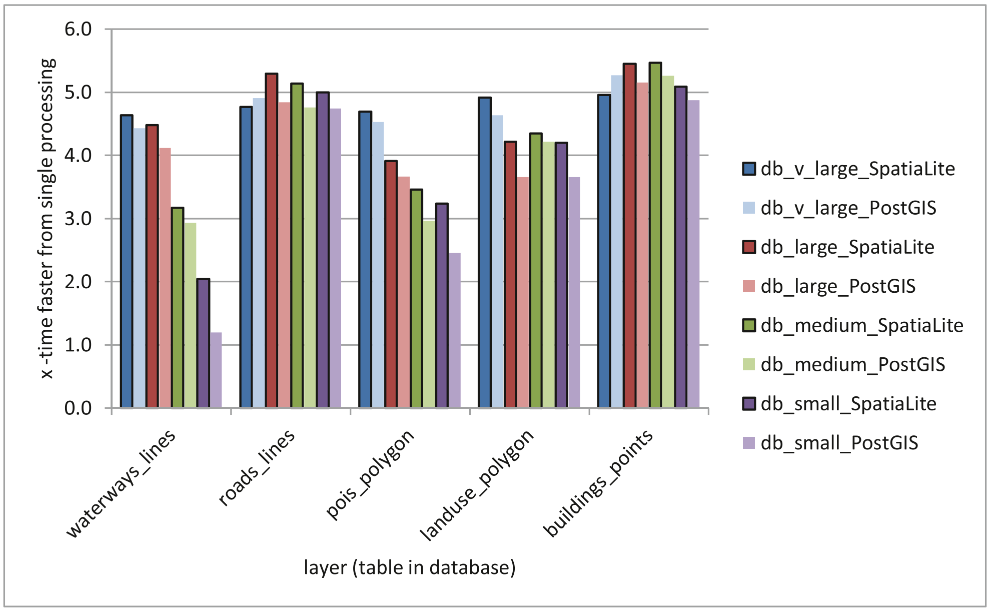

Fig. 7. Compare single-threaded and multi-threaded computation of the sum of buffer areas around all objects (in different databases). The vertical axis shows how many times multi-thread processing is faster than single-thread (computed by dividing multi-thread time by single thread time).

homogeneous; clusters of more geometrically complex objects are assigned to a single computing thread, causing them to bear a greater load. The researcher confirmed this hypothesis by observing the uneven completion of tasks by individual threads, some of which completed the task after half of the entire analysis time had elapsed. Moreover, at the end of the analysis, one thread was always most loaded with complex objects. The solution to this issue could rest in employing a different way of splitting the objects to be analyzed.

The researcher also conducted tests on an analysis based on processing a single layer to calculate the area of the buffer generated around all objects of the layer (Figs. 6 and 7). As a result of the benchmark test, the use of all six processor threads increased performance by a maximum of 5.5 times that of single processing. Guo et al. (2020) presents parallel buffer function implementation in conventional GIS platforms (e.g., ArcGIS, MapGIS), obtaining a maximum increase for six threads of about 5.5 times compared to that for a single thread. The researcher's result was the same, but, in addition to generating the buffer, his analysis also calculated its area.

Due to the increase in the number of database reading operations, whether analysis time would change significantly when using slow HDDs versus a very fast RAM disk. However, the type of disk used for the analyzed SQLite/SpatiaLite databases did not affect analysis time; perhaps employing larger databases would make it possible to show the difference between the access speed and the throughput of the storage device on which the database is saved, as differences between storage devices are evident in other applications (Xu et al., 2015).

The limitation in implementing the proposed algorithm is the amount of RAM memory available to store the result in a Python variable. If the resulting table from the SQL query is larger than the available RAM, then a slowdown will be observed, because the system starts swapping memory to disk.

PostgreSQL/PostGIS databases can be used on cloud computing (Ray et al., 2013), it is possible to adapt the algorithm to work in a distributed environment. The proposed algorithm could allow the database to be split into fewer data partitions, depending on the number of computers working in the cloud.

6. Conclusion

The article presents a method of multithread use of the SQLite/SpatiaLite and PostgreSQL/PostGIS database for spatial relationship analysis. Employing the Python programming language enabled a noticeable increase in the speed of spatial analyses. The benchmark test's results showed a 3.6X speed-up time increase on SQLite/SpatiaLite and 5.1X speed-up time increase on PostgreSQL/PostGIS over that associated with traditional single-thread processing, and, in the case of small databases (i.e., up to several thousand objects), performance decreased due to the time needed to split the data processes into individual threads on SQLite/SpatiaLite. The tests also showed that the increase in the processing speed of SQL queries on spatial relationships is not a positive, linear function of the number of computational threads used. That is, the more cores used, the smaller was the increase in productivity.

An analysis of the graphs comparing performance (Fig. 4, Table 2, Table 3) demonstrates a systematic increase in performance as the number of objects used in the analysis increased. The db_v_large database allowed for the greatest increase of the speed of parallel processing most often compared to that obtained single-thread processing. In the case of analyses performed with the use of both linear layers and lines and points, performance gain no longer increased with the use of objects from the db_large database, and the increase in the number of analyzed objects did not lead to an increase in the performance of multicore processing. In the case of the polygon-with-polygon analysis, polygon-with-point analysis, and line-with-polygon layers analysis, the increase in performance was accompanied by an increase in the number of spatial objects taking part in the analysis.

With the small SQLite/SpatiaLite database, the introduction of multithread processing led to a decrease in processing performance in almost all tests, except for touches() and crosses() analyses in pairs: polygons with polygon and line with points objects, where a small increase in performance was observed. This was due to the time required for splitting the objects to be analyzed into individual threads (preparation of the merging of object pairs and counting the number of pairs). In the case of db_large and db_v_large (large databases), the proportion of this time was small compared to the query-processing time.

A large part of the time was consumed in the preliminary calculation of records (Table 4, Appendix B) to divide the data into individual processing threads (i.e., data for the LIMIT and OFFSET commands). However, this preliminary calculation allowed the query to be broken up into separate processor threads without having to split the database.

The introduction of multithreaded processing on a small PostgreSQL/PostGIS database led to a 2.3X mean speed increase for the SQL query, and 4.5X maximum speed increase was observed on the spatial function within() using lines and polygons as well as a 4.4X speed increase on spatial function intersects(). The preliminary calculation for count() of pairs of objects in the small databases was much faster compared to SQLite/SpatiaLite. Query processing on PostgreSQL/PostGIS consumed much more time compared to the SQLite/SpatiaLite database system. For example, a polygon-with-point crosses() analysis on db_v_large consumed 32 207 [sec] using standard single-core query (on SQLite/SpatiaLite, it was 14 593 [sec]) and 6 284 [sec] using parallel (6-core) query (on SQLite/SpatiaLite 4 536 [sec]). However, the performance gains in PostgreSQL/PostGIS (single versus parallel processing) are generally greater than in SQLite/SpatiaLite.

The author, in his further work on improving multicore processing in SQL spatial queries, will improve the method for splitting the data to be analyzed into individual threads. Improving the distribution of data prepared for analysis of spatial relationships will make possible an increase in performance because it will allow better use of CPU cores throughout the analysis.

Data availability

The data that support the findings of this study are openly available in https://doi.org/10.17632/gfxfng2sc5.1 hosted at Mendeley Data (Ilba, 2020).

Computer code availability

SQLite and PostgreSQL parallel code for speed-up spatial query processing; developer: Mateusz Ilba; lab phone: (+48) 12-293-59-72, milba@uek.krakow.pl; 2020; software required: system with Python 2.7 (with installed extension: Cyqlite, Mod_spatialite, Psycopg2), PostgreSQL with PostGIS; program language: Python 2.7; program size: 3.25 KB; access: https://github.com/MateuszIlba/SpatiaLite-parallel-processing and https://github.com/MateuszIlba/PostGIS-parallel-processing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Funding: This work was supported by the subsidy granted to Cracow University of Economics, POTENTIAL Program.

Appendix A. Supplementary data

Supplementary data to this article can be found online at https://doi.org/10.1016/j.cageo.2021.104840.

References

Addair, T.G., Dodge, D.A., Walter, W.R., Ruppert, S.D., 2014. Large-scale seismic signal analysis with Hadoop. Comput. Geosci. 66, 145–154. https://doi.org/10.1016/j.cageo.2014.01.014.

Adler, D.W., 2001. DB2 spatial extender-spatial data within the RDBMS. In: Proceedings of the 27th VLDB Conference, Roma, Italy, pp. 687–690. https://dl.acm.org/doi/10.5555/645927.758383.

Agarwal, S., Rajan, K.S., 2016. Performance analysis of MongoDB versus PostGIS/PostGreSQL databases for line intersection and point containment spatial queries. Spatial Inform. Res. 24 (6), 671–677. https://doi.org/10.1007/s41324-016-0059-1.

Aji, A., Hoang, V., Wang, F., 2015. Effective Spatial Data Partitioning for Scalable Query Processing, pp. 1–12 arXiv preprint. https://arxiv.org/abs/1509.00910v1.

Aji, A., Wang, F., Saltz, J.H., 2012. Towards building a high performance spatial query system for large scale medical imaging data. In: Proceedings of the 20th International Conference on Advances in Geographic Information Systems, Redondo Beach, pp. 309–318. https://doi.org/10.1145/2424321.2424361. California, USA.

Alomari, M.A., Yusoff, M.H., Samsudin, K., Ahmad, R.B., 2019. Light database encryption design utilizing multicore processors for mobile devices. In: 2019 IEEE 15th International Colloquium on Signal Processing & its Applications (CSPA), pp. 254–259. https://doi.org/10.1109/CSPA.2019.8696084. Penang, Malaysia.

Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B., 1990. The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pp. 322–331. https://doi.org/10.1145/93597.98741. Atlantic City, New Jersey, USA.

Brinkhoff, T., Kriegel, H.P., Seeger, B., 1996. Parallel processing of spatial joins using R-trees. In: Proceedings of the Twelfth International Conference on Data Engineering, pp. 258–265. https://doi.org/10.1109/ICDE.1996.492114. New Orleans, Louisiana, USA.

Carniel, A.C., Ciferri, R.R., Ciferri, C.D., 2019. A generic and efficient framework for flash-aware spatial indexing. Inf. Syst. 82, 102–120. https://doi.org/10.1016/j.is.2018.09.004.

Cremer, S., Bagein, M., Mahmoudi, S., Manneback, P., 2017. Improving performances of an embedded relational database management system with a hybrid CPU/GPU processing engine. In: Francalanci, C., Helfert, M. (Eds.), Data Management Technologies and Applications. DATA 2016. Communications in Computer and Information Science, vol. 737. Springer, Cham, pp. 160–177. https://doi.org/10.1007/978-3-319-62911-7_9.

Dai, J., 2009. Efficient Concurrent Operations in Spatial Databases. Doctoral dissertation, Virginia Polytechnic Institute and State University. http://hdl.handle.net/10919/28987.

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113. https://doi.org/10.1145/1327452.1327492.

Di Gregorio, F., Varrazzo, D., 2020. Psycopg: PostgreSQL database adapter for Python. Retrieved January, 2021, from. https://www.psycopg.org/docs/.

Dobos, L., Szüle, J., Bodnár, T., Hanyecz, T., Sebők, T., Kondor, D., et al., 2013. A multi-terabyte relational database for geo-tagged social network data. In: 2013 IEEE 4th International Conference on Cognitive Infocommunications (CogInfoCom), pp. 289–294. https://doi.org/10.1109/CogInfoCom.2013.6719259. Budapest, Hungary.

Esri, 2019. Spatial framework for hadoop. https://github.com/Esri/spatial-framework-for-hadoop. (Accessed 26 February 2020).

Geofabrik, 2018. Geofabrik download server. Retrieved February 26, 2020, from. http://download.geofabrik.de/europe/poland.html.

Giannousis, K., Bereta, K., Karalis, N., Koubarakis, M., 2018. Distributed execution of spatial SQL queries. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 528–533. https://doi.org/10.1109/BigData.2018.8621908. Seattle, Washington, USA.

Guo, M., Han, C., Guan, Q., Huang, Y., Xie, Z., 2020. A universal parallel scheduling approach to polyline and polygon vector data buffer analysis on conventional GIS platforms. Transactions in GIS. https://doi.org/10.1111/tgis.12670.

Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp. 47–57. https://doi.org/10.1145/602259.602266. Boston, Massachusetts, USA.

Hagedorn, S., Räth, T., 2017. Efficient spatio-temporal event processing with Stark. In: International Conference on Extending Database Technology (EDBT), pp. 570–573. https://doi.org/10.5441/002/edbt.2017.72. Venice, Italy.

He, Z., Liu, G., Ma, X., Chen, Q., 2019. GeoBeam: a distributed computing framework for spatial data. Comput. Geosci. 131, 15–22. https://doi.org/10.1016/j.cageo.2019.06.003.

Hoel, E.G., Samet, H., 1994. Performance of data-parallel spatial operations. In: Proceedings of 20th International Conference on Very Large Data Bases (VLDB), pp. 156–167. Santiago, Chile.

Hu, F., Yang, C., Schnase, J.L., Duffy, D.Q., Xu, M., Bowen, M.K., Lee, T., Song, W., 2018. ClimateSpark: an in-memory distributed computing framework for big climate data analytics. Comput. Geosci. 115, 154–166. https://doi.org/10.1016/j.cageo.2018.03.011.

Ilba, M., 2020. Set of SQLite SpatiaLite Databases from OpenStreetMap Data - Poland. Mendeley Data. https://doi.org/10.17632/gfxfng2sc5.1 vol. 1.

Jin, P., Xie, X., Wang, N., Yue, L., 2015. Optimizing R-tree for flash memory. Expert Syst. Appl. 42 (10), 4676–4686. https://doi.org/10.1016/j.eswa.2015.01.011.

Kai, F., 2016. A Shared Memory-Based Lock Manager for SQLite on Multicore Platform. M.S. thesis. Department of Computer Science and Engineering, College of Engineering, Seoul National University. http://hdl.handle.net/10371/122652.

Karthi, S., Prabu, S., 2018. Improved spatial query processing framework for spatial data. Procedia Comput. Sci. 132, 759–774. https://doi.org/10.1016/j.procs.2018.05.088.

Kim, J., Kim, S.G., Nam, B., 2013. Parallel multi-dimensional range query processing with R-trees on GPU. J. Parallel Distr. Comput. 73 (8), 1195–1207. https://doi.org/10.1016/j.jpdc.2013.03.015.

Limkar, S.V., Jha, R.K., 2019. Computing over encrypted spatial data generated by IoT. Telecommun. Syst. 70 (2), 193–229. https://doi.org/10.1007/s11235-018-0479-4.

Liu, C., Ma, R., Zhang, L., 2020. Analysis of spatial indexing mechanism and its application in data management: a case study on spatialite database. In: IOP Conference Series: Earth and Environmental Science, vol. 428, pp. 1–8. https://doi.org/10.1088/1755-1315/428/1/012037, 012037.

Papadopoulos, A., Manolopoulos, Y., 2003. Parallel bulk-loading of spatial data. Parallel Comput. 29 (10), 1419–1444. https://doi.org/10.1016/j.parco.2003.05.003.

Park, K., 2014. Location-based grid-index for spatial query processing. Expert Syst. Appl. 41 (4), 1294–1300. https://doi.org/10.1016/j.eswa.2013.08.027.

Priya, M., Kalpana, R., 2018. Distributed processing of location based spatial query through vantage point transformation. Future Comput. Inform. J. 3 (2), 296–303. https://doi.org/10.1016/j.fcij.2018.09.002.

Python Documentation, 2019. Version 2.7.17, 16.6. multiprocessing — process-based "threading" interface. Retrieved February 26, 2020, from. https://docs.python.org/2.7/library/multiprocessing.html?highlight=multiprocessing#module-multiprocessing.

Ramsey, P., 2019. Parallel PostGIS and PgSQL 12. Retrieved February 26, 2020, from. http://blog.cleverelephant.ca/2019/05/parallel-postgis-4.html.

Ray, S., Simion, B., Brown, A.D., 2011. Jackpine: a benchmark to evaluate spatial database performance. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 1139–1150. https://doi.org/10.1109/ICDE.2011.5767929. Hannover, Germany.

Ray, S., Simion, B., Brown, A.D., Johnson, R., 2013. A parallel spatial data analysis infrastructure for the cloud. In: Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 284–293. https://doi.org/10.1145/2525314.2525347. New York, United States.

Real, L.C.V., Silva, B., 2018. Full speed ahead: 3D spatial database acceleration with GPUs. In: 44th International Conference on Very Large Data Bases (VLDB), pp. 1–5. Rio De Janeiro, Brazil, arXiv preprint. https://arxiv.org/abs/1808.09571v1.

Romero, O., Herrero, V., Abelló, A., Ferrarons, J., 2015. Tuning small analytics on Big Data: data partitioning and secondary indexes in the Hadoop ecosystem. Inf. Syst. 54, 336–356. https://doi.org/10.1016/j.is.2014.09.005.

Roumelis, G., Vassilakopoulos, M., Corral, A., Manolopoulos, Y., 2017. Efficient query processing on large spatial databases: a performance study. J. Syst. Software 132, 165–185. https://doi.org/10.1016/j.jss.2017.07.005.

Schön, B., Mosa, A.S.M., Laefer, D.F., Bertolotto, M., 2013. Octree-based indexing for 3D pointclouds within an Oracle spatial DBMS. Comput. Geosci. 51, 430–438. https://doi.org/10.1016/j.cageo.2012.08.021.

Shehab, E., Algergawy, A., Sarhan, A., 2017. Accelerating relational database operations using both CPU and GPU co-processor. Comput. Electr. Eng. 57, 69–80. https://doi.org/10.1016/j.compeleceng.2016.12.014.

Shvachko, K., Kuang, H., Radia, S., Chansler, R., 2010. The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. https://doi.org/10.1109/MSST.2010.5496972. Incline Village, Nevada, USA.

Simion, B., Ray, S., Brown, A.D., 2012. Speeding up spatial database query execution using gpus. Procedia Comput. Sci. 9, 1870–1879. https://doi.org/10.1016/j.procs.2012.04.205.

Solihin, W., Eastman, C., Lee, Y.C., 2017. Multiple representation approach to achieve high-performance spatial queries of 3D BIM data using a relational database. Autom. ConStruct. 81, 369–388. https://doi.org/10.1016/j.autcon.2017.03.014.

SpatiaLite, 2017a. SpatiaLite: SpatialIndex from. https://www.gaia-gis.it/fossil/libspatialite/wiki?name=SpatialIndex. (Accessed 26 February 2020).

SpatiaLite, 2017b. Dynamically Loading SpatiaLite as an Extension Module from. https://www.gaia-gis.it/fossil/libspatialite/wiki?name=mod_spatialite. (Accessed 26 February 2020).

SQLite, 2020. SQL as Understood by SQLite. Retrieved February 26, 2020, from. https://www.sqlite.org/lang_select.html#limitoffset.

Tarmur, S., Özturan, C., 2019. Parallel classification of spatial points into geographical regions. In: 18th International Symposium on Parallel and Distributed Computing. ISPDC, Amsterdam, Netherlands, pp. 9–15. https://doi.org/10.1109/ISPDC.2019.000-3.

Vinhas, L., De Souza, R.C.M., Câmara, G., 2003. Image data handling in spatial databases. In: GeoInfo 2003, Campos Do Jordão, Brazil.

Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., et al., 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In: Proceedings of the 8th ACM International Systems and Storage Conference, pp. 1–11. https://doi.org/10.1145/2757667.2757684. Haifa, Israel.

Yao, X., Mokbel, M.F., Alarabi, L., Eldawy, A., Yang, J., Yun, W., et al., 2017. Spatial coding-based approach for partitioning big spatial data in Hadoop. Comput. Geosci. 106, 60–67. https://doi.org/10.1016/j.cageo.2017.05.014.

You, S., Zhang, J., Gruenwald, L., 2015. Large-scale spatial join query processing in cloud. In: 2015 31st IEEE International Conference on Data Engineering Workshops, pp. 34–41. https://doi.org/10.1109/ICDEW.2015.7129541. Seoul, South Korea.

Zhang, J., You, S., 2012. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In: Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, pp. 23–32. https://doi.org/10.1145/2447481.2447485. Redondo Beach, California, USA.

Zhang, J., You, S., Gruenwald, L., 2014. Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs. Inf. Syst. 44, 134–154. https://doi.org/10.1016/j.is.2014.01.005.

Zhou, X., Abel, D.J., Truffet, D., 1998. Data partitioning for parallel spatial join processing. GeoInformatica 2 (2), 175–204. https://doi.org/10.1023/A:1009755931056.