High-Performance Partition-based and Broadcast-based Spatial Join on GPU-Accelerated Clusters

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY, USA
syou@gc.cuny.edu

Jianting Zhang
Dept. of Computer Science
The City College of New York
New York, NY, USA
jzhang@cs.ccny.cuny.edu

Le Gruenwald
Dept. of Computer Science
The University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

***Abstract*—The rapid growing volumes of spatial data have brought significant challenges on developing high-performance spatial data processing techniques in parallel and distributed computing environments. Spatial joins are important data management techniques in gaining insights from large-scale geospatial data. While several distributed spatial join techniques based on symmetric spatial partitions have been implemented on top of existing Big Data systems, they are not capable of natively exploiting massively data parallel computing power provided by modern commodity Graphics Processing Units (GPUs). In this study, we have extended our distributed spatial join framework that was originally designed for broadcast-based spatial joins to partition-based spatial joins. Different from broadcast-based spatial joins that require one input side of a spatial join to be a point dataset and the other side to be sufficiently small for broadcast, the new extension supports non-point spatial data on both sides of a spatial join and allows them to be both large in volumes while still benefit from native parallel hardware acceleration for high performance. We empirically evaluate the performance of the proposed partition-based spatial join prototype system on both a workstation and Amazon EC2 GPU-accelerated clusters and demonstrate its high performance when comparing with the state-of-the-art. Our experiment results also empirically quantify the tradeoffs between partition-based and broadcast-based spatial joins by using real data.**

Keywords—Spatial Join, Partition-based, Broadcast-based, GPU, Distributed Computing

I. INTRODUCTION

Advances of sensing, modeling and navigation technologies and newly emerging applications, such as satellite imagery for Earth observation, environmental modeling for climate change studies and GPS data for location dependent services, have generated large volumes of geospatial data. Very often multiple spatial datasets need to be joined to derive new information to support decision making. For example, for each pickup location of a taxi trip record, a spatial join can find the census block that it falls within. Time-varying statistics on the taxi trips originate and designate at the census blocks can potentially reveal travel and traffic patterns that are useful for city and traffic planning. As another example, for each polygon boundary of a US Census Bureau TIGER record, a polyline intersection based spatial join can find the river network (or *linearwater*) segments that it intersects. While traditional Spatial Databases and Geographical Information System (GIS) have provided decent supports for small datasets, the performance is not acceptable when the data volumes are large. It is thus desirable to use Cloud computing to speed up spatial join query processing in computer clusters. As spatial joins are typically both data intensive and computing intensive and Cloud computing facilities are increasingly equipped with modern multi-core CPUs, many-core Graphics Processing Units (GPUs) and large memory capacity[i], new Cloud computing techniques that are capable of effectively utilizing modern parallel and distributed platforms are both technically preferable and practically useful.

Several pioneering Cloud-based spatial data management systems, such as HadoopGIS [1] and SpatialHadoop [2], have been developed on top of the Hadoop platform and have achieved impressive scalability. More recent developments, such as SpatialSpark [3], ISP-MC+ and ISP-GPU [4], are built on top of in-memory systems, such as Apache Spark [5] and Cloudera Impala [6], respectively, with demonstrable efficiency and scalability. We refer to Section II for more discussion on the distributed spatial join techniques and the respective research prototype systems. Different from HadoopGIS and SpatialHadoop that perform spatial partitioning before globally and locally joining spatial data items, i.e., partition-based spatial join, ISP-MC+ and ISP-GPU are largely designed for broadcast-based spatial join, where one side (assuming the right side) dataset in a spatial join is broadcast to the partitions of another side (assuming the left side) which is a point dataset (not necessarily spatially partitioned) for local joins. While SpatialSpark supports both broadcast-based and partition-based spatial join (in-memory), when both sides of a spatial join are large in volumes, broadcast-based spatial joins require significantly more memory capacity and is more prone to failures (e.g., due to the out of memory issue). This makes partition-based spatial join on SpatialSpark a more robust choice. We note that partition-based spatial joins typically require reorganizing data according to partitions on either external storage (HDFS for HadoopGIS and SpatialHadoop) or in-memory (SpatialSpark) through additional steps.

Our previous work has demonstrated that SpatialSpark is significantly faster than HadoopGIS and SpatialHadoop for partition-based spatial joins [7]. ISP-MC+ and ISP-GPU, which have exploited native multi-core CPU and GPU parallel computing power, are additionally much faster than SpatialSpark for broadcast-based spatial join [8]. They both use point-in-polygon test as the spatial joining criteria. The results bring an interesting question on the achievable speedups of distributed partition-based spatial joins over the established Hadoop-based spatial join techniques represented by SpatialHadoop when parallel hardware is natively exploited. Unfortunately, the underlying platform of ISP-MC+ and ISP-GPU, i.e., Impala, cannot be easily extended to support partition-based spatial joins.

In this study, we aim at developing a partition-based spatial join technique on top of the LDE engine that we have developed previously [8] for distributed large-scale spatial data processing and evaluating its performance using real world datasets. We have developed efficient designs and implementations on both multi-core CPUs and GPUs for polyline intersection based spatial joins. To the best of our knowledge, the reported work is the first to design an efficient parallel polyline intersection algorithm on GPUs and GPU-accelerated clusters for partition-based spatial joins. As demonstrated in the experiment section, the new technique significantly outperforms implementations on CPUs using open source geometry libraries, including GEOS [ii] used by HadoopGIS [1] and JTS[iii] used by SpatialHadoop [2] and SpatialSpark [3]. By comparing with existing systems such as SpatialHadoop using publically available large-scale geospatial datasets (CENSUS TIGER and USGS *linearwater*), we demonstrate that our techniques that are capable of natively exploit parallel computing power of GPU-accelerated clusters can achieve significant higher performance, due to the improvements of both parallel geometry library for polyline intersection tests and distributed computing infrastructure for in-memory processing. Furthermore, for the first time, we directly compare the end-to-end performance of partition-based spatial join techniques with that of broadcast-based spatial join techniques in a typical parallel and distributed computing environment in Cloud also using publically available datasets. Experiments on joining NYC 2013 taxi point dataset and Census block polygon dataset have revealed that broadcast-based spatial join can be several times more performant than partition-based spatial join. GPS point locations are among the fastest growing spatial data and joining points with administrative zones and urban infrastructure data represented as polyline and polygonal datasets are getting increasingly popular. As such, developing specialized broadcast spatial join techniques to maximize end-to-end performance can be more preferred than generic partition-based spatial join techniques that have been provided by existing Spatial Big Data systems.

The rest of the paper is organized as follows. Section II provides background, motivation and related work. Section III introduces broadcast-based and partition-based spatial joins and their implications in distributed spatial join query processing. Section IV is the system architecture and design and implementation details for partition-based spatial joins. Section V reports experiments and their results. Finally Section VI is the conclusion and future work directions.

II. BACKGROUND AND MOTIVATION

Spatial join is a well-studied topic in Spatial Databases and we refer to [9] for an excellent survey in traditional serial, parallel and distributed computing settings. A spatial join typically has two phases, i.e., the filtering phase and the refinement phase. The filtering phase pairs up spatial data items based on their Minimum Bounding Rectangle (MBR) approximation by using either pre-built or on-the-fly constructed spatial index. The refinement phase applies computational geometry algorithms to filter out pairs that do not satisfy the required spatial criteria in the spatial join, typically in the form of a spatial predicate, such as point-in-polygon test or polyline intersection test. While most of the existing spatial join techniques and software implementations are based on serial computing on a single computing node, techniques for parallel and distributed spatial joins have been proposed in the past few decades for different architectures [9]. Although these techniques differ significantly, a parallel spatial join typically has additional steps to partition input spatial datasets (spatially and non-spatially) and join the partitions globally (i.e., global join) before data items in partition pairs are joined locally (i.e., local join).

As Hadoop-based Cloud computing platforms become mainstream, several research prototypes have been developed to support spatial data management, including spatial joins, on Hadoop. HadoopGIS [1] and SpatialHadoop [2] are among the leading works on supporting spatial data management by extending Hadoop. We have also extended Apache Spark for spatial joins and developed SpatialSpark [3], which has achieved significantly higher performance than both HadoopGIS and SpatialHadoop [7]. HadoopGIS adopts the Hadoop Streaming [iv] framework and uses additional MapReduce jobs to shuffle data items that are spatially close to each other into the same partitions before a final MapReduce job is launched to process re-organized data items in the partitions. SpatialHadoop extends Hadoop at a lower level and has random accesses to both raw and derived data stored in the Hadoop Distributed File System (HDFS[v]). By extending *FileInputFormat* defined by the Hadoop runtime library, SpatialHadoop is able to spatially index input datasets, explicitly access the resulting index structures stored in HDFS and query the indexes to pair up partitions based on the index structures, before a *Map*-only job is launched to process the pairs of partitions in distributed computing nodes. SpatialSpark is based on Apache Spark. Spark provides an excellent development platform by automatically distributing tasks to computing nodes, as long as developers can express their applications as data parallel operations on collection/vector data structures, i.e., Resilient Distributed Datasets (RDDs) [5]. The automatic distribution is based on the key-value pairs of RDDs which largely separate domain logic from parallelization and/or distribution. A more detailed review of the three research prototype systems and their performance comparisons using public datasets are reported in [7].

Different from SpatialHadoop, HadoopGIS and SpatialSpark are forced to access data sequentially within data partitions due to the restrictions of the underlying platform (Hadoop Streaming for HadoopGIS and Spark RDD for SpatialSpark). The restrictions, due to the streaming data model (for HadoopGIS) and Scala functional programming language (for SpatialSpark), have significantly lower the capabilities of the two systems in efficiently supporting spatial indexing and indexed query processing. Indeed, spatial indexing in the two systems is limited to intra-partitions and requires on-the-fly reconstructions from raw data. The implementations of spatial joins on two datasets are conceptually cumbersome as partition boundary is invisible and cross-partition data reference is supported by neither Hadoop Streaming nor Spark RDD. To solve the problem, both HadoopGIS and SpatialSpark require additional steps to globally pair up partitions based on spatial intersections before parallel/distributed local joins on individual partitions. While the additional steps in SpatialSpark are implemented as two *GroupBy* primitives in Spark which are efficient for in-memory processing, they have to be implemented as multiple MapReduce jobs in HadoopGIS and significant data movements across distributed computing nodes (including Map, Reduce and Shuffle phases) are unavoidable. The excessive disk I/Os are very expensive and largely contribute to HadoopGIS's lowest performance among the three systems. On the other hand, while SpatialHadoop has support on storing, accessing and indexing geometric data in binary formats with random access capabilities by significantly extending Hadoop runtime library, its performance is significantly limited by Hadoop and is inferior to SpatialSpark for data-intensive applications, largely due to the performance gap between disk and memory. Note that we have deferred the discussions on spatial partitioning in the three systems to Section III.

As both HadoopGIS and SpatialHadoop are based on Hadoop and SpatialSpark is based on Spark, which are either based on Java or Scala programming language and they all rely on Java Virtual Machine (JVM), native parallel programming tools which are likely to help achieve higher performance cannot be easily incorporated. Furthermore, currently JVM supports Single Instruction Multiple Data (SIMD) computing power on neither multi-core CPUs nor GPUs. Although HadoopGIS has attempted to integrate GPUs into its MapReduce/Hadoop framework, the performance gain was not significant [10]. To effectively utilize the increasingly important SIMD computing power, we have extended the leading open source in-memory Big Data system Impala [6] that has a C++ backend to support spatial joins using native parallel processing tools, including OpenMP [vi], Intel Threading Building Blocks (TBB [vii]) and Compute Unified Device Architecture (CUDA[viii]). By extending block-based join in Impala, our In-memory Spatial Processing (ISP) framework [4] is able to accept a spatial join in a SQL statement, parse the data in the two sides of a spatial join in chunks (row-batches), build spatial index on-the-fly to speed up local joins in chunks. While ISP employs the SQL query interface inherited from Impala, its current architecture also limits our extension to broadcast-based spatial join, i.e., broadcasting the whole dataset on the right side of a join to chunks of the left side of the join for local join (termed as left-partition-right-broadcast). As Impala is designed for relational data, in order to support spatial data under the architecture, ISP was forced to represent geometry as strings in the Well-Know-Text (WKT[ix]) format. In a way similar to HadoopGIS, this increases not only data volumes (and hence disk I/Os) significantly, but also infrastructure overheads. The simple reason is that text needs to be parsed before geometry can be used and intermediate binary geometry needs to be converted to strings for outputting.

Our recent work on developing the Lightweight Distributed Execution (LDE) engine for spatial joins aims at overcoming these disadvantages by allowing accessing HDFS files (including both data and index) randomly in a principled way [8]. Experiments have demonstrated the efficiency of LDE-MC+ and LDE-GPU when compared with ISP-MC+ and ISP-GPU, respectively. Note that ISP-MC adopts the GEOS geometry library and uses OpenMP for intra-node parallelization. On the other hand, ISP-MC+, ISP-GPU, LDE-MC+ and LDE-GPU all use our columnar layout and custom geometry routines for point-in-polygon test, point-to-polyline distance computation and polyline-to-polyline intersection test. Both techniques have contributed significantly to the higher performance of the prototype systems with respect to end-to-end runtime, in addition to utilizing parallel and distributed computing units, as reported previously. We note that HadoopGIS also uses GEOS library within the wrapped reducer for local join as ISP-MC does. However, we found that the C++ based GEOS library is typically several times slower than its Java counterpart (JTS library) which makes ISP-MC unattractive when comparing ISP-MC with SpatialSpark [3]. We suspect that this is also one of the important factors that contributes to SpatialHaoop's superior performance when comparing the end-to-end performance of HadoopGIS and SpatialHadoop reported in [7].

The developments of ISP and LDE are driven by the practical needs of spatially joining GPS point locations with urban infrastructure data or global ecological zone data based on several spatial join criteria, including point-in-polygon test and point-to-polyline distance. This makes broadcast-based spatial join a natural choice in ISP based on Impala as broadcast is natively supported by Impala. However, different from HadoopGIS and SpatialHadoop that naturally support partition-based spatial join based on MapReduce and Hadoop, supporting partition-based spatial join based on Impala is nontrivial, despite several efforts were attempted. Fortunately, our in-house developed LDE engine, which is conceptually equivalent to Hadoop runtime, allows an easier extension to support partition-based spatial joins, although it is initially developed as a succession of ISP to support broadcast-based spatial join. We next introduce both partition-based and broadcast-based spatial joins in Section 3 as they were originally developed in their respective systems before we present the new design of partition-based spatial join in LDE in Section 4.

III. DISTRIBUTED SPATIAL JOIN USING BROADCAST- AND PARTITION- BASED METHODS

To successfully join large-scale geospatial datasets, especially when the volumes of input datasets or intermediate data exceed the capacity of a single machine, efficient distributed spatial join techniques that are capable of effectively utilized aggregated resources across multiple computing nodes are essential. When both datasets in a spatial join are large in volumes (or "big" for short), as introduced previously, a common practice is to spatially partition both input datasets, which can be any spatial data types, for parallel and/or distributed spatial joins on the partitions. We term the scenario as partition-based spatial join on symmetric spatial datasets, or *symmetric spatial join* for short. While the scenario is generic and seems universally applicable, in the context of parallel and distributed computing, it requires data reorganization on both sides so that data items that belong to the same partition can be stored in the same computing node to avoid random data accesses across multiple nodes in distributed memory or distribute file systems, which are very expensive. An additional disadvantage is that, the reorganized data items lose their original orderings and it might be equally or even more costly to rearrange the joined results based on the original ordering of either side. The order-keeping spatial joins may be desirable in many applications, e.g., assigning a polyline/polygon identify to a point based on 1-to-1 mapping spatial relationship. For the scenario in these applications, while spatial-partition based techniques are still applicable due to their generality, the excessive and unnecessary data movements might prevent them from achieving optimum performance. Given that point datasets in such spatial joins are typically much larger than polylines/polygons that they are joining in quantities and we can apply broadcast-based techniques for the spatial joins more efficiently, we term the scenario as broadcast-based spatial join on asymmetric spatial datasets, or a*symmetric spatial join* for short. We next provides more discussions on the implications of the distinctions of these two categories of spatial joins in parallel and distributed computing settings.

A. *Spatial Partition-based Spatial Join*

In partition-based spatial join, if neither side is indexed, an on-demand partition schema can be created and both sides are partitioned on-the-fly according to on-demand schema. This approach has been used in previous works [1,2,11]. On the other hand, when both datasets have already been partitioned according to a specific partition schema, partitions can be matched as pairs and each pair can be processed independently [2]. The quality of partitions has significant impact on the efficiency of parallel spatial join technique on multiple computing nodes. First of all, a high quality spatial partition schema minimizes expensive intra-node communication cost. Second, such a schema can also minimize the effects from stragglers (slow nodes), which typically dominate the end-to-end performance. Therefore, parallel spatial join techniques typically divide a spatial join task into small (nearly) equal-sized and independent subtasks and process those small tasks in parallel efficiently. The left side of Fig. 1 illustrates on-demand partition and the right side of the figure shows the process of matching already partitioned datasets. The two input datasets (A and B) are matched as partition pairs using either on-demand partition schema (left) or matching existing partitions (right). Here A and B are represented as partitions where each partition contains a subset of the original dataset, e.g., A_1 and B_1 are in partition 1. Partition pairs are subsequently assigned to computing nodes and each node performs local spatial join processing.

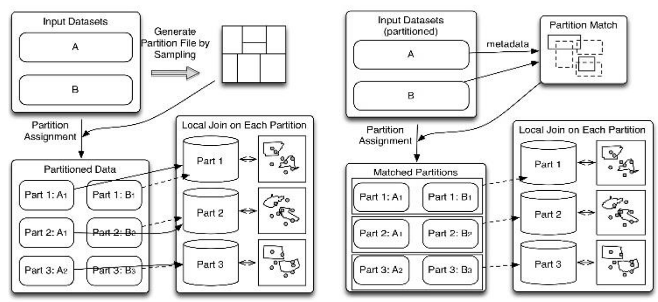

Figure 1 Spatial Partition-based Spatial Join

Several spatial partition schemas have been proposed previously, such as fixed-grid partition (FGP), Binary Split Partition (BSP) and Sort-Tile Partition (STP). Figure 2 shows spatial partition results based on FGP (top-left), BSP (top-right) and STP (bottom) of Census Block data in the New York City (NYC), respectively. Although we have empirically chosen STP for spatial partition in this study as the resulting partitions typically have better load balancing features, we refer to [11] for brief introductions to other spatial partition schemas. We note that our system can accommodate all spatial partition schemas as long as spatial data items in a partitioning dataset are assigned to partitions in a mutually exclusive and collectively exhaustive manner. In STP (bottom part of Figure 2), data is first sorted along one dimension and split into equal-sized strips. Within each strip, final partitions are generated by sorting and splitting data according to the other dimension. The parameters for STP are the number of splits at each dimension as well as a sampling ratio. Different from BSP, STP sorts data at most twice and do not need recursive decompositions, which is more efficient for large datasets based on our experiences. We also note that our decision on adopting STP as the primary spatial partition schema agrees with the finding in SpatialHadoop [2].

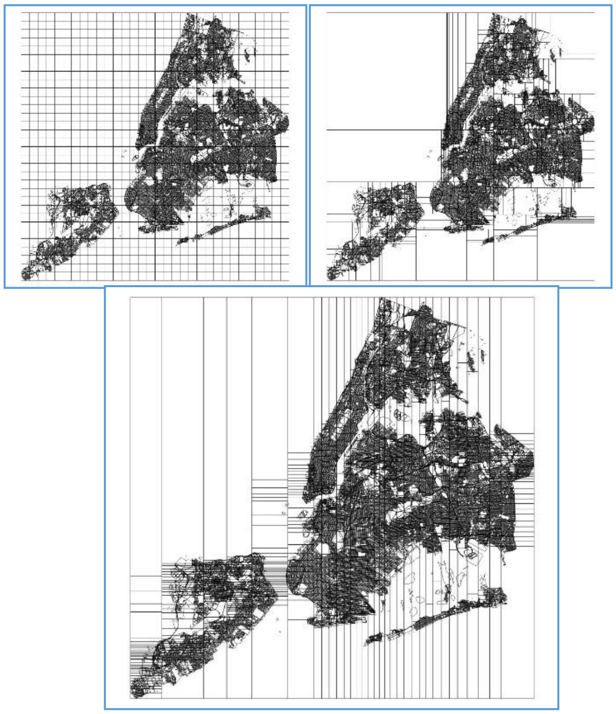

Figure 2 Spatial Partitions of NYC Census Block Data using FGP (top-left), BSP (top-right) and STP (bottom)

B. Broadcast-based Spatial Join

As discussed previously, broadcast-based spatial join (illustrated in Figure 3) can be considered as a generality-efficiency tradeoff with partition-based spatial join for spatial joins with one side (assuming the left side) of the inputs be a point dataset and the other side (assuming the right side) of the inputs be small in data volume. The left side point data input can be partitioned based on its storage order while the right side can be broadcast to all partitions of the left side for local spatial joins. Clearly, in this scenario, neither side requires data reorganization. This is desirable in distributed computing as data movements in distributed file systems are known to be expensive. Furthermore, as a point on the left side typically is joined with at most one data item on the right side in the scenario, the join result can be represented as a list of identifies of data items on the right side. The list of identifies can be stored as a data column in a distributed file system separately, which could be much more efficient than concatenating multiple columns from both input dataset of a spatial join and write the join result to a distributed file system. Note that the correspondence between the original points and the identifier list is based on the implicit ordering of data item positions.

Figure 3 shows an example of broadcast-based spatial join, where the right side is bulk-loaded using R-tree and broadcast to all computing nodes, and, the left side is divided into chunks where each chunk is processed by a processing unit (computing node). Broadcast-based spatial join typically works as follows. The first step is to broadcast the small dataset to all computing nodes; an optional on-demand spatial index may be created during the broadcasting. As a result, each node owns a copy of the small dataset as well as the spatial index if applicable, and they will be persistent in memory for efficient accesses. In the second step, the left side is divided into equal-sized chunks based on their positions in the sequence, i.e., the i^{th} data item is assigned to partition i/B where B is the chunk size, and we term it as sequence-based partition.

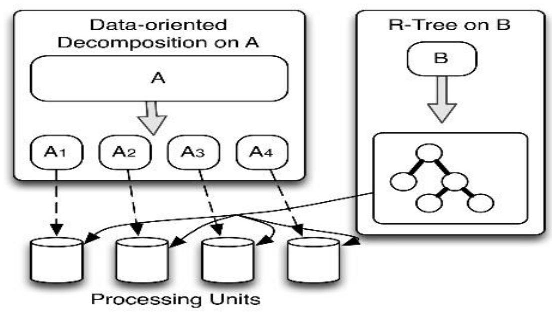

Figure 3 Broadcast-based Spatial Join

Compared with space-based partition in partition-based spatial joins, sequence-based partition is much simpler. As broadcast mechanism is typically supported in Big Data systems for relational data (such as Cloudera Impala as well as Apache Spark), it is relatively easy to extend existing Big Data systems to support broadcast-based spatial joins as demonstrated in our ISP prototypes on top of Impala [3, 4]. In the local join phase, a partition of the left side input will be joined with a broadcast copy of the right side input. To speed up local joins, while the partition is loaded from a distributed file system (such as HDFS), spatial index can be constructed with negligible overhead as disk I/O typically dominates. Spatial index of the right side input can be constructed either before broadcast or after broadcast. While broadcast the spatial index together with the right side input may reduce the overhead of computing nodes to build the index individually, it will complicate the broadcast process and the benefit may not always justify the complexity. The spatial index of the right side input can also be pre-built and stored in HDFS, which can be read by all computing nodes when needed (i.e., disk-based broadcast by HDFS). While this approach may eliminate index construction overhead for the right side dataset to be broadcast in real time, it will incur additional disk I/O time and complicate system design as well. The choices need to be carefully analyzed and justified. Our ISP prototypes construct spatial index for the local copy of the right side in real time as it is very difficult to access custom index files from HDFS within Impala. On the other hand, our LDE prototype reads both the right side input and its index from HDFS as accesses to index files in LDE are built-in. We refer to the respective publications for more details on the designs and implementations of broadcast-based spatial join techniques.

On the downside, when the right side of a spatial join is large in volume, broadcast-based join will incur significant memory overheads, which is linear with respect to data volume of the right side input. This necessitates partition-based spatial joins by spatially partitioning the right side that is too big to broadcast. We next move to Section IV to present the details of partition-based spatial join on GPUs and GPU-accelerated clusters, which is one of our main contributions of this study.

IV. SYSTEM ARCHITECTURE AND IMPLEMENTATIONS

The distributed partition-based spatial join technique is developed on top of the LDE engine we have developed previously. While designed to be lightweight, the LDE engine supports asynchronous data transfer over network, asynchronous disk I/O and asynchronous computing and we refer to [8] for details. The asynchronous design makes the join processing in a non-blocking manner, which can deliver high performance by hiding latency from disk access and network communication. In this study, the LDE architecture is extended in several aspects to accommodate partition-based spatial joins. The overall system architecture is illustrated in the left side of Figure 4. First, the master node reads partition information from HDFS and populates the task schedule queue associated with the scheduler. This is different from the original LDE design for broadcast-based spatial join where the queue task is populated by sequence-based partitions that are computed on-the-fly. Second, batches are now computed from partition pairs, instead of from partitions of the left side input (point dataset). Subsequently, an optimization technique has been developed to minimize duplicated HDFS disk I/O within batches. We note that a batch is the basic unit for distributed processing in LDE. The details of task scheduling, batch dispatching and processing are provided in Section IV.A. Second, different from previous studies that rely on open source geometry libraries to perform spatial refinement (polyline-intersection in particular in this study) which only work on CPUs in a serial computing setting, we have developed data parallel algorithms for the spatial refinement that are efficient on both GPUs and multi-core CPUs. The details of the key technical contribution in this study are provided in Section IV.B. We note that, similar to SpatialHadoop, currently spatial partitioning of input datasets is treated as a preprocessing step and therefore it is not shown in Figure 4. By following a few simple naming and formatting conventions, the partitions can be computed either serially, in parallel or imported from third party packages such as SpatialHadoop, which make the architecture flexible.

A. Scheduling, Dispatching and Processing

Step 1 in Figure 4 pairs partitions of both sides of inputs to compute partition pairs. As the numbers of spatial partitions of the two input datasets in a spatial join are typically small, pairing the partitions based on their MBRs is fast using any in-memory algorithms on a single computing node. We thus implement the step at the master node. Assuming both datasets are partitioned and partition boundaries are stored in HDFS, the index loader of the master node in LDE loads partition MBRs of both datasets from HDFS before performing in-memory parallel pair matching to generate a partition pair list in Step 1 as shown in the lower-left side of Figure 4. The details of Step 1 are further illustrated in the top-right part of Figure 4. The matched partition pairs are then pushed into a task queue which will be dispatched by the scheduler at the master node in Step 2. While it is intuitive to use a pair as a batch for distributed execution in a worker node, however, such design will generate a large number of batches that require substantial communication overheads, which will

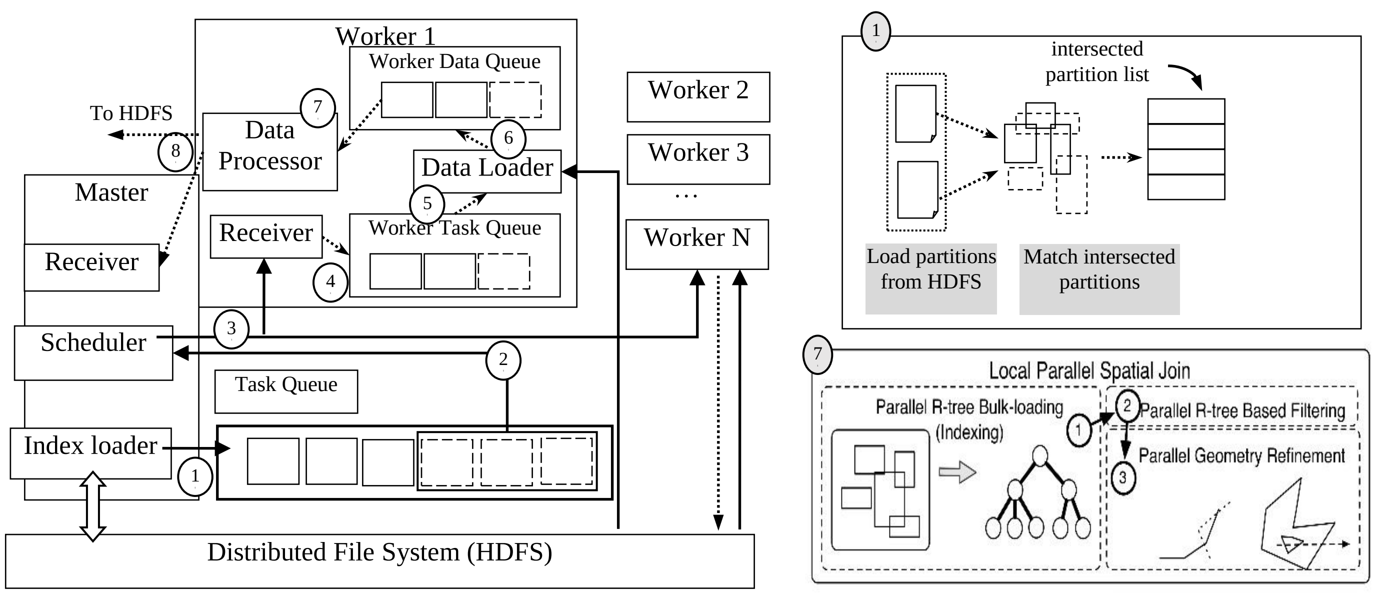

Figure 4 System Architecture and Modules

negatively impact system performance. In addition, such a naïve scheduling policy will also result in hardware underutilization on worker nodes, especially for GPUs. Unless a matched partition pair involves large numbers of data items in the input datasets, the workload for processing a single pair is unlikely to saturate GPUs that have large numbers of processing units.

In order to reduce network bandwidth pressure and minimize overhead of hardware accelerators, we divide the list of matched pairs into multiple batches where each batch contains multiple partition pairs which are processed at a time on a worker node. The number of partition pairs in a batch is determined by the processing capability of the worker node, e.g., memory capacity and available processing units. When a worker node joins the master node during the system initialization, the processing capability information is sent to the master node. Based on such information, the size of a batch can be determined. Unlike distributed join in SpatialHadoop that each partition pair is processed by a Mapper in a MapReduce job and requires Hadoop runtime to do the scheduling which is ignorant to spatial data, our system spatial conscious.

Workload dispatching is achieved by maintaining a receiver thread in each worker node (Step 3). The receiver thread listens to a socket port dedicated for the LDE framework. Note that in each batch only data locations and offsets to the partitions are stored and transferred between the master node and a worker node. This is because all worker nodes are able to access data directly through HDFS and the data accesses are likely to be local due to HDFS file block replications. As such, the network communication overhead of transferring batches is likely to be very low in our system. Received batches are pushed into a task queue of the worker node (Step 4). A separate thread of the worker node is designated to load data from HDFS for each batch at worker node (Step 5). The data loaded for each partition pair is kept in an in-memory data queue which will be processed next (Step 6).

Since a partition from one dataset may overlap with multiple partitions from the other dataset, there will be duplicated IO requests for the same partition. To reduce the redundant IO requests, we sort the partition pairs in the master node before they are formed as batches. The duplicated partitions will then appear consecutively in the partition pair list. As a result, duplicated partitions for a batch can be detected and only one copy of the duplicated partitions is loaded from HDFS at a worker node. This improvement significantly reduces expensive data accesses to HDFS. The local spatial join logic is fully implemented in Step 7 and the join query results are written to HDFS in Step 8. While we refer to the left side of Figure 4 for the workflow of Steps 1-8, the details of Step 7 are further illustrated in the lower-right part of Figure 4 and will be explained in details next.

B. Data Parallel Local Spatial Join on GPUs

As introduced previously, each worker node has a local parallel spatial join module to process partition pairs in a batch. In addition to the designs and implementations for point-in-polygon test based spatial joins and point-to-polyline distance based spatial joins that we have reported previously, in this study, we aim at designing and implementing a new spatial predicate for polyline intersection test that can be integrated into the classic *filter-and-refinement* spatial join framework [9] on GPUs. We reuse our GPU-based R-tree technique [12] for spatial filtering and we next focus on the polyline intersection test spatial predicate.

```
Input: polyline representations, candidate pairs
Output: intersection status
Polyline_Intersection:
(pid, qid) = get_polyline_pair(block_id)
(p_start, p_end) = linestring_offset(pid)
(q_start, q_end) = linestring_offset(qid)
__shared__ intersected = False
for p_linestring from p_start to p_end
    for q_linestring from q_start to q_end
        __syncthreads()
        workload = len(p_linestring) * len(q_linestring)
        processed  = 0
        while (!intersected && processed < workload)
          if (thread_id + processed >= workload) continue
           p_seg = get_segment(p_linestring, thread_id)
           q_seg = get_segment(q_linestring, thread_id)
           is_intersect = segment_intersect(p_seg, q_seg)
           if (is_intersect) intersected = True
           processed += num_threads_per_block
           __syncthreads()
        end while
        if (intersected)
           results[block_id] = True
           return
    end for //p_linestring
end for //q_linestring
```

Figure 5 Polyline Intersection Kernel

Assuming R-Tree based spatial filtering generates a list of polyline pairs where the MBRs of the polylines in each pair intersect. Given two polylines, $P(p_1, p_2, \ldots, p_m)$ and $Q(q_1, q_2, \ldots, q_n)$, where p and q are line segments, we can perform intersection test on P and Q by checking whether any of two line segments in P and Q intersect. As we have a list of candidate pairs, it is intuitive to assign each pair to a processing unit for parallel processing. However, the numbers of line segments of polylines can be very different among pairs which may results in poor performance when the naïve parallelization approach is applied to GPUs due to unbalanced workloads across GPU threads. Perhaps more importantly, as each GPU thread needs to loop through all line segments of another polyline in a matched pair separately in the naïve parallelization, neighboring GPU threads are not likely to access the same or neighboring memory locations. The non-coalesced memory accesses may lower the GPU performance significantly as accesses to GPU memory can be more than an order of magnitude slower than accesses to CPU memory, given the quite different cache configurations on typical GPUs and CPUs.

As balanced workload and coalesced global memory access are crucial in exploiting the parallel computing power on GPUs, we have developed an efficient data-parallel design on GPUs to achieve high performance. First, we minimize unbalanced workload by applying parallelization at line segment level rather than at polyline level. Second, we maximize coalesced global memory accesses by laying out line segments from the same polyline consecutively on GPU memory and letting each GPU thread process a line segment. Figure 5 lists the kernel of the data parallel design of polyline intersection test on GPUs and more details are explained below.

In our deign, each pair of polyline intersection test is assigned to a GPU computing block to utilize GPU hardware scheduling capability to avoid unbalanced workload created by variable polyline sizes. Within a computing block, all threads are used to check line segments intersection in parallel. Since each thread performs intersection test on two line segments where each segment has exactly two endpoints, the workload within a computing block is perfectly balanced. The actual implementation for real polyline data is a little more complex as a polyline may contain multiple linestrings which may not be continuous in polyline vertex array. The problem can be solved by recording the offsets of the linestrings in the vertex array of a polyline and using the offsets to locate vertices of line segments of the linestrings to be tested.

Line 1-3 of the kernel in Figure 5 retrieve the positions of non-continuous linestrings, followed by two iterations in Line 5 and 6. For each pair of linestrings, all threads of a block retrieve line segments in pairs and test for intersection (Line 10-17). We designate a shared variable, *intersected*, to indicate whether there is any pair of line segments intersected for the polyline pair. Once a segment pair intersects, the *intersected* variable is set to true and becomes visible to all threads within the thread block. The whole thread block then immediately terminates (Line 18-20). When the thread block returns, GPU hardware scheduler can schedule another polyline pair on a new thread block. Since there is no synchronization among thread blocks, there will be no penalty even though unbalanced workloads are assigned to blocks.

Figure 6 illustrates the design of polyline intersection for both multi-core CPUs and GPUs. After the filter phase, candidate pairs are generated based on MBRs of polylines. As we mentioned previously, a pair of polylines can be assigned either to a CPU thread (multi-core CPU implementation) for iterative processing by looping though line segments or to a GPU thread block (GPU implementation) for parallel processing. While GPUs typically have hardware schedulers to automatically schedule multiple thread blocks on a GPU, explicit parallelization on polylines across multiple CPU cores is needed. While we use OpenMP with dynamic scheduling for the purpose in this study, other parallel libraries on multi-core CPUs, such as Intel TBB, may achieve better performance by utilizing more complex load balancing algorithms. In both multi-core CPU and GPU implementations, we have exploited native parallel programming tools to achieve higher performance based a shared-memory parallel computing model. This is different from executing Mapper functions in Hadoop where each Mapper function is assigned to a CPU core and no resource sharing is allowed among CPU cores.

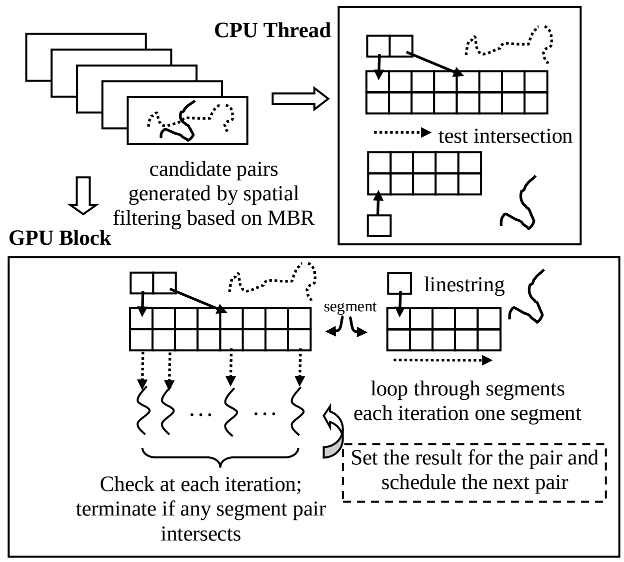

Figure 6 Data Parallel Polyline Intersection Design

V. EXPERIMENTS AND RESULTS

A. *Experiment Setup*

In order to conduct performance study, we have prepared real world datasets for two experiments which are also publically accessible to facilitate independent evaluations on different parallel and/or distributed platforms. The first experiment is designed to evaluate point-in-polygon test based spatial join, which uses pickup locations from New York City taxi trip data in 2013[x] (referred as *taxi*) and New York City census blocks[xi] (referred as *nycb*). The second experiment is designed to evaluate polyline intersection based spatial join using two datasets provided by SpatialHadoop [xii] , namely TIGER *edge* and USGS *linearwater*. For all four datasets, only geometries are used from original datasets for the experiment purpose and the specifications are listed in Table 1. We also apply appropriate preprocessing on the datasets for running on different systems. For SpatialHadoop, we use its R-tree indexing module and leave other parameters by default. For our system, all the datasets are partitioned using Sort-Tile partition (256 tiles for *taxi*, 16 tiles for *nycb*, 256 tiles for both *edge* and *linearwater*) for partitioned-based spatial joins. Note that datasets do not need pre-partition for broadcast-based spatial joins.

We have prepared several hardware configurations for experiment purposes. The first configuration is a single node cluster with a workstation that has dual 8 core CPUs at 2.6 GHz (16 physical cores in total) and 128 GB memory. The large memory capacity makes it possible to experiment spatial joins that require significant amount of memory. The workstation is also equipped with an Nvidia GTX Titan GPU with 2,688 cores and 6GB memory. Another configuration is a 10-node Amazon EC2 cluster, in which each node is a *g2.2xlarge* instance consists of 8 vCPUs and 15 GB memory, is used to for scalability test. Each EC2 instance has an Nvidia GPU with 1,568 cores and 4GB memory. We vary the number of nodes for scalability test and term the configurations as EC2-X where X denotes the number of nodes in the cluster. Both clusters are installed with Cloudera CDH-5.2.0 to run SpatialHadoop (version 2.3) and SpatialSpark (with Spark version 1.1).

Table 1 Experiment Dataset Sizes and Volumes

Dataset	# of Records	Size
Taxi	169720892	6.9GB
Nycb	38839	19MB
Linearwater	5857442	8.4GB
Edge	72729686	23.8GB

B. Results of Polyline Intersection Performance on Standalone Machines

We first evaluate our polyline intersection designs using *edge* and *linearwater* datasets on both multi-core CPUs and GPUs on our workstation and a *g2.2xlarge* instance without involving distributed computing infrastructure. As the polyline intersection time dominates the end-to-end time in this experiment, the performance can be used to evaluate the efficiency of the proposed polygon intersection technique on both multi-core CPUs and GPUs. The results are plotted in Figure 7, where *CPU-Thread* and *GPU-Block* refer the implementations of the proposed design, i.e., assigning a matched polyline pair to a CPU thread and a GPU computing block, respectively. Note the data transfer time between CPUs and GPUs are included when reporting GPU performance.

For GPU-Block, the approximately 50% higher performance on the workstation than on the single EC2 instance shown in Figure 7 represents a combined effect of about 75% more GPU cores and comparable memory bandwidth when comparing the GPU on the workstation and the EC2 instance. For CPU-Thread, the 2.4X better performance on the workstation than that on the EC2 instance reflect the facts that the workstation has 16 CPU cores while the EC2 instance has 8 virtualized CPUs, in addition to Cloud virtualization overheads. While the GPU is only able to achieve 20% higher performance than CPUs on our high-end workstation, the results show 2.6X speedup on the EC2 instance where both the CPU the GPU are less powerful. Note that the reported low GPU speedup on the workstation represents the high efficiency of our polygon intersection test technique on both CPUs and GPUs. While it is not our intension to compare our data parallel polygon intersection test implementations with those that have been implemented in GEOS and JTS, we have observed orders of magnitude of speedups. As reported in the next subsection, the high efficiency of the geometry API actually is the key source for our system to significantly outperform SpatialHadoop for partition-based spatial joins where SpatialHadoop uses JTS for geometry APIs.

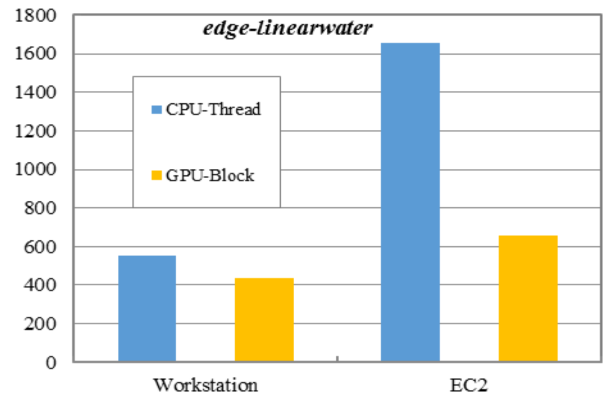

Figure 7 Polyline Intersection Performance (in seconds) in edge-linearwater experiment

C. Results of Distributed Partition-Based Spatial Joins

The end-to-end runtimes (in seconds) for the two experiments (*taxi-nycb* and *edge-linearwater*) under the four configurations (WS, EC2-10, EC2-8 and EC2-6) on the three systems (SpatialHadoop, LDE-MC+ and LDE-GPU) are listed in Table 2. The workstation (denoted as WS) here is configured as a single-node cluster and is subjected to distributed infrastructure overheads. LDE-MC+ and LDE-GPU denote the proposed distributed computing system using multi-core CPUs and GPUs, respectively. The runtimes of the three systems include spatial join times only and the indexing time for the two input datasets are excluded. The *taxi-nycb* experiment uses point-in-polygon test based spatial join and the *edge-linearwater* uses polyline intersection base spatial join.

From Table 2 we can see that, comparing with SpatialHadoop, the LDE implementations on both multi-core CPUs and GPUs are at least an order of magnitude faster for all configurations. The efficiency is due to several factors. First, the specialized LDE framework is a C++ based implementation which can be more efficient than general purpose JVM based frameworks such as Hadoop (on which SpatialHadoop is based). The in-memory processing of LDE is also an import factor where Hadoop is mainly a disk-based system. With in-memory processing, intermediate results do not need to write to external disks which is very expensive. Second, as mentioned earlier, the dedicated local parallel spatial join module can fully exploit parallel and SIMD computing power within a single computing node. Our data-parallel designs in the module, including both spatial filter and refinement steps, can effective utilize current generation of hardware, including multi-core CPUs and GPUs. From a scalability perspective, the LDE engine has achieved reasonable scalability. When the number of EC2 instances is increased from 6 to 10 (1.67X), the speedups vary from 1.39X to 1.64X. The GPU implementations can further achieve 2-3X speedups over the multi-core CPU implementations which is desirable for clusters equipped with low profile CPUs.

Table 2 Partition-based Spatial Join Runtimes (s)

		WS	EC2-10	EC2-8	EC2-6
taxi-nycb	SpatialHadoop	1950	1282	1315	2099
	LDE-MC+	191	39	50	63
	LDE-GPU	111	19	23	30
edge-linearwater	SpatialHadoop	9887	3886	5613	6915
	LDE-MC+	554	219	260	360
	LDE-GPU	437	97	114	135

D. Results of Broadcast-Based Distributed Spatial Joins

In addition to comparing the performance of the partition-based spatial join among SpatialHadoop, LDE-MC+ and LDE-GPU in both *taxi-nycb* and *edge-linearwater* experiments, we have also compared the performance of broadcast-based spatial join with partition-based spatial join using the *taxi-nycb* experiment. The *edge-linearwater* cannot use broadcast-based join due to memory constraint as discussed earlier. From the results presented in Table 3, it can be that the LDE framework outperforms all other systems including ISP, which is expected due to the lightweight infrastructure overhead by design.

Table 3 Broadcast-based Spatial Join Runtimes (s)

		WS	EC2-10	EC2-8	EC2-6
taxi-nycb	SpatialSpark	355	101	108	144
	ISP-MC+	130	36	44	54
	ISP-GPU	96	21	27	34
	LDE-MC+	119	22	25	31
	LDE-GPU	50	12	15	16

Comparing the runtimes in Table 3 with Table 2 for the same taxi-nycb experiment, we can observe that the broadcast-based spatial join is much faster (up to 2X) than partition-based spatial join using the LDE engine, even without including the overhead of preprocessing in partition-based spatial join. The results support our discussions in Section 3. This may suggest that broadcast-based spatial join should be preferred whereas possible. When native parallelization tools are not available, the broadcast-based spatial join implemented in SpatialSpark can be an attractive alternative, which outperforms SpatialHadoop by 5.5X under WS configuration and 12-14.5X under the three EC2 configurations.

VI. CONCLUSIONS AND FUTURE WORK

In this study, we have designed and implemented partition-based spatial join on top of our lightweight distributed processing engine. By integrating distributed processing and parallel spatial join techniques on GPUs within a single node, our proposed system can perform large-scale spatial join effectively and achieve much higher performance than the state-of-the-art. Experiments comparing the performance of partition-based and broadcast-based spatial joins suggest that broadcast-based spatial join techniques can be more efficient when joining a point dataset and a relatively small spatial dataset that is suitable for broadcast.

As for future work, we plan to further improve single node local parallel spatial module by adding more spatial operators with efficient data-parallel designs. We also plan to develop a scheduling optimizer for the system that can perform selectivity estimation to help dynamic scheduling to achieve higher performance.

ACKNOWLEDGEMENT

This work is supported through NSF Grants IIS-1302423 and IIS-1302439.

REFERENCES

1. A. Aji, F. Wang, et al (2013). Hadoop-gis: A high performance spatial data warehousing system over mapreduce. In *VLDB, 6(11)*, pages 1009–1020.
2. E. Eldawy and M. F. Mokbel (2015). SpatialHadoop: A MapReduce Framework for Spatial Data. In Proc. IEEE ICDE'15.
3. S. You, J. Zhang and L. Gruenwald (2015). Large-Scale Spatial Join Query Processing in Cloud. In Proc. IEEE CloudDM'15.
4. S. You, J. Zhang and L. Gruenwald (2015). Scalable and Efficient Spatial Data Management on Multi-Core CPU and GPU Clusters: A Preliminary Implementation based on Impala, in Proc. IEEE HardBD'15.
5. M. Zaharia, M. Chowdhury et al (2010). Spark: Cluster Computing with Working Sets. In Proc. HotCloud.
6. M. Kornacker and et al. (2015). Impala: A modern, open-source sql engine for hadoop. In Proc. *CIDR*'15.
7. S. You, J. Zhang and L. Gruenwald (2015). You, J. Zhang and L. Gruenwald (2015). Spatial Join Query Processing in Cloud: Analyzing Design Choices and Performance Comparisons. To appear in Proc. IEEE HPC4BD. Online at http://www-cs.ccny.cuny.edu/~jzhang/papers/sjc_compare_tr.pdf
8. J. Zhang, S. You and L. Gruenwald (2015). A Lightweight Distributed Execution Engine for Large-Scale Spatial Join Query Processing. In Proc. IEEE Big Data Congress'15.
9. E. H. Jacox and H. Samet (2007). Spatial Join Techniques. ACM Trans. Database Syst., vol. 32, no. 1, p. Article #7.
10. A. Aji, G. Teodoro and F. Wang (2014). Haggis: turbocharge a MapReduce based spatial data warehousing system with GPU engine. In Proc. ACM BigSpatial'14.
11. H. Vo, A. Aji and F. Wang (2014). SATO: a spatial data partitioning framework for scalable query processing. In Proc. ACMGIS'14.
12. J. Zhang and S. You (2012). Speeding up large-scale point-in-polygon test based spatial join on GPUs. In Proc. ACM BigSpatial, 23-32.

[i] http://aws.amazon.com/ec2/instance-types/
[ii] http://trac.osgeo.org/geos/
[iii] http://www.vividsolutions.com/jts/JTSHome.htm
[iv] http://hadoop.apache.org/docs/r1.2.1/streaming.html
[v] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[vi] http://openmp.org/wp/
[vii] https://www.threadingbuildingblocks.org/
[viii] http://www.nvidia.com/object/cuda_home_new.html
[ix] https://en.wikipedia.org/wiki/Well-known_text
[x] http://chriswhong.com/open-data/foil_nyc_taxi/
[xi] http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml
[xii] http://spatialhadoop.cs.umn.edu/datasets.html