Lightweight Distributed Execution Engine for Large-Scale Spatial Join Query Processing

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

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

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

Abstract— **Existing Big Data systems are mostly designed for relational data. They are either incapable or inefficient in processing large-scale semi-structured data efficiently due to the inherent limitations on data abstraction, indexing support and exposure to native parallel programming tools. In this study, we report our work in developing a lightweight distributed execution engine for spatial join query processing on large-scale geospatial data. By integrating data parallel designs for single computing nodes, our execution engine is able to automatically dispatch data partitions to distributed computing nodes for efficient local execution on multi-core CPUs and GPUs. The execution engine supports asynchronous data transfer over network, asynchronous disk I/O and asynchronous computing. It also directly accesses distributed file systems to support creating and using indices conveniently and efficiently. In addition to be lightweight by design, which has less than 1,000 Lines Of Code (LOC), experiments using a real world application have demonstrated significant efficiency improvement over our previous works on extending a leading in-memory Big Data system (Impala) for spatial join query processing.**

Keywords—Lightweight, Distributed Computing, Execution Engine, Spatial Join

I. Introduction

Advanced sensors, such as GPS devices and smartphones, have generated huge amounts of location data. Very often point location data need to be joined with urban infrastructure data, such as road networks and administrative zones, to better understand human mobility patterns and, subsequently, improve urban planning, traffic control and infrastructure maintenance. As an example, thousands of New York City (NYC) buses send back GPS locations every 30 seconds[1] which amount to millions of points daily and billions yearly. The Global Biodiversity Information Facility (GBIF[2]) has accumulated more than 500 million species occurrence records each of which is often associated with a (latitude, longitude) pair. It is essential to map the occurrence records to various ecological regions to understand the biodiversity patterns and make conservation plans. Such applications require spatial join query processing, a well-defined problem in spatial databases research [1] [2]. This functionality has been provided by major commercial and open source spatial database systems as well as Geographical Information Systems (GIS). However, the amounts of spatial data in these applications (e.g., in the order of tens of millions to billions of data items) well exceed the processing capabilities of traditional disk-resident systems running on single computing nodes and, thus, require new systems to reduce processing times, from days or hours to minutes or even seconds, in order to be practically useful for researchers and decision makers [3].

Our previous efforts in extending the leading in-memory Big Data system Impala [4] to support spatial data processing with a focus on spatial joins on multi-core CPU and Graphics Processing Units (GPU) equipped clusters have achieved limited success [5] [6]. There are several major issues in the “extension” oriented approach. First, the data model supported by Impala is mostly designed for relational data, which makes it inefficient to represent and manipulate semi-structured data, such as spatial data. To support spatial data in Impala, we were forced to represent geometry of spatial data as strings. This not only increases data volume significantly (and hence disk I/O overheads and memory footprint) but also requires converting geometry between text and binary back and forth, which is not efficient. Second, the tightly integrated end-to-end system makes it difficult (if not impossible) to plug-in new sophisticated indexing techniques to support spatial joins. It becomes practically intractable when one attempts to modify existing Big Data systems (such as Impala) with hundreds of thousands of lines of code. Third, our experiments have shown up to 50% infrastructure overhead when running the extended system for spatial joins on a single computing node.

We therefore consider it desirable to develop a lightweight distributed computing system that is capable of supporting indexing and query processing on spatial data in an easy and natural way. In this paper, we report our work on developing a Lightweight Distributed Execution (LDE) engine for large-scale spatial data processing with an initial focus on spatial joins. Our work is unique in the following aspects that are not available in existing systems: 1) It is lightweight and easy to extend, 2) It supports spatial data as well as other types of non-relational data, and 3) it is C++ based and supports Single Instruction Multiple Data (SIMD [7]) computing on both multi-core CPUs and GPUs for high performance.

For the rest of the paper, we first introduce the background and related work in Section II. Section III presents the system architecture of LDE and its implementation details. Section IV provides experiment results on large-scale point-in-polygon test based spatial join query processing. Finally Section V is the conclusion and future work directions.

II. Background, Motivation and Related Work

Assuming that the two datasets in a spatial join have been indexed, the first step in a parallel and/or distributed implementation of the spatial join, called spatial filtering [2], is to pair up partitions in the two datasets based on spatial relationships (e.g., intersection). Subsequently, the geometry of data items within the paired spatial partitions are tested based on the desired spatial predicates, e.g., point-in-polygon test and whether the distance between a point and a polyline is within a certain threshold. It is natural to distribute the paired partitions to worker nodes for distributed processing. It can be seen that spatial indices can potentially reduce the complexity of the spatial join from O(M*N) to O(M*log(N) or even O(M), where M and N are the numbers of records in the two datasets of a spatial join. Numerous spatial indexing and spatial join query processing techniques [1] [2] have been proposed in the past four decades and some of them have been implemented in mainstream spatial database systems. Not surprisingly, the majority of the techniques are designed for serial computing on uniprocessors in disk-resident systems, which significantly limits performance on processing large-scale geospatial data.

The last few years have seen dramatic new developments in Big Data systems after Hadoop/MapReduce became the mainstream software framework for distributed processing of Big Data. Several open source software, such as Cloudera Impala [4], Apache Spark [8], AsterixDB [9] and Stratosphere [10], in addition to commercial products, have been developed to improve Hadoop with respect to functionality, efficiency and usability. However, although there are Big Data systems that are designed to support graph data, most of the existing mainstream Big Data systems are designed for relational data [11] [12] [13]. These Big Data systems typically adopt a simplified data model that supports only element-wise operations in a streaming mode, such as scans and aggregations on tuples [12], vertex or edges [14], where indexing is not actively exploited. Although sophisticated indexing techniques have played an important role in improving the efficiency of both traditional single-node databases and distributed databases, existing Big Data systems generally lack a systematic framework to support creating and using indexes to efficiently support more complex operations [15]. Unfortunately, many operations on semi-structured data, such as spatial data, trajectory data and time series data, require global indexing to pair up partitions in multiple datasets before the pairs can be dispatched to distributed computing nodes for local processing.

Given the complexity of developing full-fledged Big Data systems, which typically consist hundreds of thousands of Lines of Code (LOC), a natural way to support new operations of semi-structured data is to extend existing Big Data systems instead of developing a new system from scratch. For spatial data, HadoopGIS [16] and SpatialHadoop [17] are among the leading works on supporting spatial data management within the Hadoop ecosystem. HadoopGIS adopts the Hadoop Streaming[3] framework and uses additional MapReduce jobs to shuffle data items that are spatially close to each other into 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[4]). 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. Compared with the Hadoop Streaming framework adopted in HadoopGIS which allows only strictly sequential data accesses, being able to manipulate HDFS files through system level APIs and access disk files randomly in SpatialHadoop has made it the most efficient spatial data management system on top of Hadoop (to the best of our knowledge). Several research prototypes for spatial operations have been built on top of SpatialHadoop (such as GISQF [18]). However, like all Hadoop-based systems, excessive disk I/Os in executing Hadoop jobs is a performance bottleneck when compared with in-memory systems, such as Impala and Spark.

We have extended both Impala and Spark for spatial joins and developed several prototype systems on them, i.e., SpatialSpark, ISP-MC, ISP-MC+ and ISP-GPU, respectively. We refer to [5] [6] for more implementation details and their performance evaluations on both a single computing node and Amazon EC2 clusters. 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) [8]. The automatic distribution is based on the key-value pairs of RDDs which largely separate domain logic from parallelization/distribution. While SpatialSpark has achieved significantly higher performance than both HadoopGIS and SpatialHadoop, the Scala language and Java Virtual Machine (JVM[5]) based Spark platform currently does not allow using SIMD computing power easily and effectively. Hadoop/Spark supports Single Instruction Multiple Data (SIMD) computing on neither multi-core CPUs nor Graphics Processing Units (GPUs). As SIMD width is getting larger, which is 4-16 way for Vector Processing Units on multi-core CPUs and 32 way on Nvidia GPUs, it is much more desirable to exploit SIMD computing power on modern hardware [7].

Impala, on the other hand, has a C++ backend which makes it possible to exploit SIMD computing power on both multi-core CPUs (using Vector Processing Units – VPUs [7]) and GPUs. While Impala is able to utilize Streaming SIMD Extensions 4 (SSE4) on Intel CPUs to significantly speed up string related operations, which may largely contribute to its superior performance when compared with Hadoop and Spark[6], SIMD instruction sets have not been used for other operations (including spatial) in Impala. Our previous works on developing ISP-MC [5], ISP-MC+ [6] and ISP-GPU [6] have demonstrated both feasibility and reasonable efficiency. Note that although the three prototype systems share the In-

Memory Spatial Processing (ISP) framework we have developed on top of Impala with spatial extensions, ISP-MC is designed for multi-core CPUs using traditional geometry library (i.e., GEOS[7]), ISP-MC+ is designed for multi-core CPUs using our own geometry library with columnar data layout [3] and ISP-GPU is designed for GPUs also with columnar data layout. ISP-GPU largely offloads spatial indexing, spatial filtering and spatial refinement to GPUs. Using our own geometry library with columnar data layout on GPUs has achieved the highest efficiency. However, tremendous technical challenges have been encountered during the journey of extending Impala for spatial data, mostly due to data model and architecture mismatches. Experiments also have shown that the overheads of supporting spatial operations in the Impala-based prototypes are high, especially for ISP-GPUs where floating point computing has been significantly sped up by GPUs.

There are several efforts in developing high-performance execution engines for distributed computing and some have been demonstrated favorably when compared with Hadoop. In particular, Sphere [19] is a C/C++ based programming framework that processes data managed by a distributed file system called Sector. Sphere manages both data and associated metadata using operating system level socket programming APIs directly and processes data in place whenever possible. Despite that Sphere has a much smaller codebase than Hadoop, experiments have demonstrated several times higher efficiency due to the simplified design and efficient implementations. AJIRA [20] is designed to be a lightweight distributed middleware for MapReduce and stream processing. It is implemented on top of a message-passing library called IRIS for distributed data communication. Different from Hadoop, each computing node in AJIRA runs multiple processes in a single multi-threaded application which removes the need of slow inter-process communication within the node. DataMPI [21] represents a unique hybrid solution by bridging the MapReduce programming model and Message Passing Interface (MPI[8]) based distributed data communication libraries. While experiments have demonstrated good performance when compared with Hadoop, a drawback is that MPI software stacks are typically complex and heavyweight. Although the library overheads may not be significant in high-end cluster computing facilities equipped with fast CPUs and high-performance interconnection networks (e.g., Infiniband[9]), its advantages are less clear on commodity clusters (e.g., Amazon EC2) when compared with distributed data communications directly on socket APIs (such as Sphere and AJIRA). The additional mandatory data movement overheads between Hadoop (Java-based) and MPI libraries (C/C++ based) are also a performance concern. SciDB[10] , an open-source analytical database system designed for array data, uses the asynchronous data communication module in the boost C++ libraries (i.e., Boost.Asio [11]) for distributed data communications in the units of array chunks. Although the performance of distributed query processing on array data has not been formally evaluated and published, data-aware (array chunks) asynchronous communication among distributed computing nodes in SciDB is conceptually similar to the Akka[12] module used in Apache Spark and could be more efficient than fine-grained synchronous communication. While these distributed systems have achieved high performance and/or scalability for both Cloud computing and High-Performance computing facilities, to the best of our knowledge, they are generally oblivious to data query semantics and do not natively support indexing in an extensible way. Nevertheless, many of these works have demonstrated that lightweight implementation does not necessarily result in low performance. On the contrary, lightweight implementations provide opportunities to significantly reduce system overheads and increase performance, in addition to high usability.

Dandelion [23], a research prototype developed by Microsoft Research Silicon Valley, aims at providing several desirable features that are similar to our work at a higher level, including language integrated query compilation, supporting runtime GPU acceleration and automatic distribution of computing tasks based on data-parallel semantics. The underlying new distributed dataflow engine of Dandelion called Moxie is similar to our work in many aspects. Unfortunately, neither Dandelion nor Moxie is open source at present. Lacking implementation details has prevented a direct comparison. GeMTC [24] shares similar goals as Dandelion with respect to automatically parallelizing and distributing tasks to computing nodes equipped with GPUs based on the scripts written in a parallel dataflow language (Swift). However, GeMTC is geared more towards traditional high-performance computing facilities for computing intensive applications (e.g., Molecular Dynamics simulations). Similar to distributed systems discussed previously, neither Dandelion nor GeMTC supports multi-dimensional indexing, although GPUs are supported in the two systems. None of these systems supports spatial indexing and query processing on either multi-core CPU clusters or GPU accelerated clusters, which makes our work unique.

III. SYSTEM ARCHITECTURE AND IMPLEMENTATIONS

We model query processing as a multi-step distributed computing job, which may include indexing of one or more input datasets and evaluating sub-clauses of a SQL query statement. During the process of evaluating a sub-clause, which may be recursive, one or more index structures may be utilized to pair up data items. We are in the process of developing a SQL parsing frontend to transform a spatial join SQL statement into a Directed Acyclic Graph (DAG) with each node representing a processing step. While we have successfully extended the Impala frontend to support several spatial predicates [5] [6], e.g., point-in-polygon test and nearest neighbor search within a distance, the new frontend aims at providing higher level optimizations by better understanding the input datasets and generating efficient indexing structures with low overheads. The developments of the SQL frontend will be reported elsewhere and we assume a spatial join query job is provided as multiple processing steps in this study. We next present the system architecture of the distributed execution engine before introducing the spatial join query processing technique for the execution engine (referred to as the LDE engine). We note that, although we currently focus on spatial data, the same design and implementation can be applied to other data types. Domain-specific indexing/query processing modules can be plugged into LDE engine easily.

A. System Architecture and Components

At each processing step, as shown in Fig. 1, our execution engine consists of a master node and a set of worker nodes. The master node may first query an in-memory repository (e.g., Zero-Hop Distributed Hashtable –ZHT [25]) or a distributed file system (e.g., HDFS) to retrieve the metadata of the input (Step 1). Based on the metadata and pre-defined rules/policies, the master node subsequently makes a schedule, parallelizes the computing in the step into multiple independent tasks and pushes the tasks into its queue (Step 2). When the master node is initialized, it launches two threads, one as a sender and one as a receiver. The sender thread checks the task queue of the master node periodically and sends tasks to available worker nodes for distributed processing (Step 3). For each available worker node, the sender launches a thread to connect the worker node and transmit data asynchronously. The receiver thread, on the other hand, listens to a socket (at a predefined port) and launches a thread to receive the processing result of a worker node and update the availability of the worker node (Step 10). The sender and receiver at the master node are coordinated through a conditional variable to ensure that tasks are only dispatched to worker nodes that are available to take tasks.

A worker node also maintains a task queue with a configurable capacity. The capacity of the task queue is advertised to the master node when the worker node registers with the master node. The worker task queue is initially filled by the master node (Step 3) based on the advertised capacity. Upon successfully completing a task, the worker node signals the master node to send a new task (Step 10). The receiver at the work node receives the task and pushes it back to the worker task queue (Step 4). A separate data loader thread periodically checks the status of the task queue and pops up a task when the task queue is not empty (Step 5). Based on the task specification, the data load thread loads relevant data from a distributed file system (e.g., HDFS) and puts the data in an in-memory data queue (Step 6). Another data processing thread periodically checks the status of the data queue and performs processing logic (to be detailed shortly) on the ready data (Step 7). When the process completes (Step 8), a sender thread at the worker node is launched to connect with the master node and send the results across the network (Step 9). Similar to the design in Step 2, two conditional variables are used in Step 5 and Step 7, respectively. The first conditional variable is used to coordinate the receiver thread and the data loader thread. The second conditional variable is used to coordinate the data loader thread and the data processer thread.

There are three queues (and their associated conditional variables) in our system architecture to support asynchronous data transfer over the network (master node), asynchronous disk I/O (worker node) and asynchronous computing (worker node), respectively. By overlapping network data transfer, disk I/O and computing, the overall system performance can be improved. By setting the tasks at proper granularities, the thread coordination overheads in accessing the concurrent queue structures can be amortized. Our implementation uses concurrent data structures, conditional variables and threads supported by Boost libraries [13], which significantly reduces the development complexity. Furthermore, our implementation uses Apache Thrift[14] to define data types and service interfaces to be shared by both the master node and worker nodes. Invoking service interfaces, instead of programming sockets directly, is not only simpler and less error-prone but also efficient. Our micro benchmarks have shown that distributed data communications by invoking service interfaces can achieve comparable performance with low-level socket programming, provided that data is transferred in sufficiently large chunks, which holds for our data parallel applications in most cases.

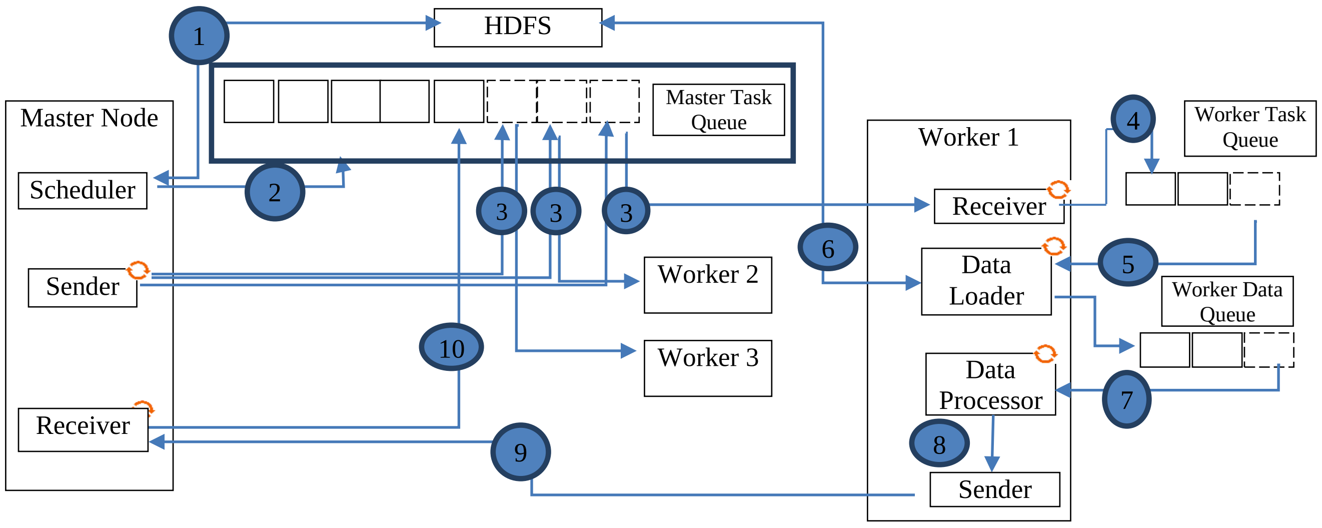


Fig. 1 LDE System Architecture and Components

From a distributed computing perspective, our approach is similar to Impala which also uses Apache Thrift and Boost concurrent data structures, but is different from Spark which uses Akka actors for distributed coordination through asynchronous messages and Netty[15] for bulk data transfer. However, compared with Impala and Spark, our implementation is lightweight indeed as the distributed computing framework (excluding spatial join logic) currently has less than 1,000 LOC, which is easy to understand, use and maintain. While Impala uses an ad-hoc hashing function and Spark adopts a round-robin policy to assign data partitions to worker nodes without considering their heterogeneity, our design/implementation has associated a weight (representing processing power) with each worker node; tasks can thus be dispatched to worker nodes based on their processing power. We note that, similar to Spark (but different from Impala), different master nodes can be chosen for multiple steps in a job to balance the workload among the distributed computing nodes, as a computing node hosting a master node typically has a heavier workload especially when the computing node also serves as a worker node.

As the LDE engine allows random accesses to HDFS files, we have chosen to adopt a column-oriented data layout design. While this is a natural extension to the column-oriented data layout in single-node platform [3] [26], distributed file systems like HDFS can make it easy to access desired data chunks in a transparent way. The design is significantly different from the designs of our SpatialSpark and ISP-based prototype systems [5] [6], where geometry is stored in the Well-Known Text (WKT[16]) format due to the restrictions imposed by the underlying platforms (Impala and Spark, respectively) as they support only relational data types. Given the partition boundaries of geometry data that are stored as binary arrays, the corresponding data can be streamed from HDFS to CPU memory and GPU memory without parsing text and rebuilding in-memory data structures, which is efficient from a practical perspective. As HDFS takes care of mapping logical data chunks to physical storage and retrieving data chunks from either local files or remote files (when necessary), it significantly reduces the complexity of application code. Since HDFS typically replicates data blocks to multiple nodes, it is likely that disk data accesses will be local which can significantly reduce network bandwidth contention and improve overall performance. The design has a different strategy than Impala (and hence our ISP variations) which relies on the Hive[17] metastore to provide the mapping between replicated data blocks and nodes and uses only nodes that have a local data block. Given that spatial joins are both data intensive and computing intensive, retrieving a relatively small amount of data from remote nodes (through HDFS) makes it possible to overlap computing with network data transfer. This may actually bring higher performance, as all computing nodes can be fully utilized in our prototype system.

The C/C++ native implementation in our prototype system also makes it possible to exploit parallel computing power using native tools on both multi-core CPUs and GPUs more easily. Different from JVM-based distributed platforms that typically treat multiple CPU cores as separate processing units, our design aims at exposing multi-core CPU and many-core GPU parallel hardware to data processing applications so that native parallel programming tools (including programming languages and runtime systems) can be used to develop efficient parallel code and achieve high performance. For each task to be processed at a worker node (c.f. the top-right part of Fig. 1), computing can be realized on multi-core CPUs, GPUs or their combinations. In particular, our GPU implementations are based on CUDA[18] and the Thrust parallel library[19] that comes with the CUDA SDK on Nvidia GPUs. Our multi-core CPU implementations are based on OpenMP[20] directives for automatic parallelization and Intel Thread Building Block (TBB[21]) for custom parallelization. For GPU programs that are developed purely based on Thrust, the library allows to compile them to multi-core CPU executables.

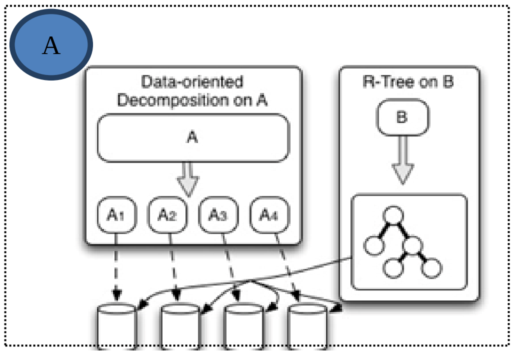
B. *Parallel and Distributed Spatial Indexing and Query Processing*

While the distributed execution engine is designed to be generic to support multiple semi-structured data types, we focus on spatial data in this study. We next introduce spatial indexing and query processing techniques for the LDE engine.

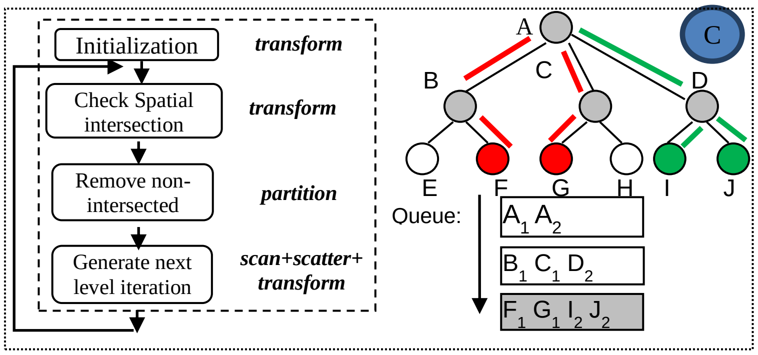
Consider the case of aligning species distribution locations (points) with ecological regions (polygons) as an intermediate step to count the number of species occurrences whose occurrence locations fall within the ecological regions. The ecological regions (referred to as polygon data), while complex (each polygon may have multiple rings and each ring may have multiple vertices), typically have a moderate data volume. On the other hand, the points (referred to as point data), although simple (the geometry of a point involves only x and y coordinates), may be large in quantity. While it is possible to partition both the point data and the polygon data using traditional spatial partition based distributed spatial join techniques, such as Partition-based Spatial Merge join (PBSM [2], illustrated in Fig. 3B), we propose to use a new design by partitioning the point data and broadcasting the polygon data for spatial join query processing, which we refer to as broadcast-based spatial join (illustrated in Fig. 2A). Without losing generality, we assume the point data and the polygon data are the left side and the right side of a spatial join, respectively. One of the advantages of the "left-partition and right-broadcast" spatial join technique is to avoid the overhead of data reordering in spatial join techniques that spatially partition both sides, which is very expensive due to large volume of the point data. Instead of spatially partitioning the point data, our technique partitions points based on their natural storage order which does not need data reordering in HDFS. As the right side is broadcast to all worker nodes, each worker node can process a partition of points by querying them against the broadcast polygons concurrently (Fig. 2B).

Based on this idea, our technique uses the partitions of point data as tasks to initialize the master task queue in the execution engine. Each worker node also reads the polygon dataset from HDFS (c.f. top of Fig. 1), which essentially achieves the purpose of polygon data broadcasting. A partition of the point dataset and a broadcast copy of the polygon dataset will be used as the two inputs of the spatial query processing on a worker node in the distributed execution engine (Step 7 in Fig. 1). As shown in Fig. 2C, we re-use our data parallel batched R-Tree query processing technique [26] , which can be compiled to both multi-core CPU and GPU code, to pair up points and the Minimum Bounding Boxes (MBBs) of polygons (i.e., spatial filtering for spatial join query processing [2]). In the spatial filtering phase of a point-in-polygon test based spatial joins, since a MBB is used to approximate a polygon and MBBs may overlap even the polygons they represent do not overlap, a point may be paired up with multiple polygons. The false positives need to be pruned in the spatial refinement phase [2] of a spatial join query processing. Given a list of (point, polygon) candidate pairs derived from the spatial filtering phase, we use OpenMP and Intel TBB for parallelization on multi-core CPUs and Thrust for parallelization on GPUs. As briefly illustrated in Fig. 3D, the list is first divided into segments where each segment is processed by a CPU thread or a GPU thread block. While a CPU thread iteratively loops through all the point-polygon pairs to decide whether the point falls within the polygon in the pair (i.e., explicit temporal loop), a GPU thread block may process multiple point-in-polygon tests simultaneously where the assignments among point-polygon pairs and GPU cores are automatically managed by GPU hardware. Different from CPUs that exploit multi-level large caches to reduce latency and improve throughputs, GPUs rely on launching a large number of concurrent threads to hide high memory access latencies while achieving excellent throughputs. The implementation of parallel spatial refinement relies on the columnar data layout discussion previously and shares a similar codebase with ISP-MC+ and ISP-GPU [6].

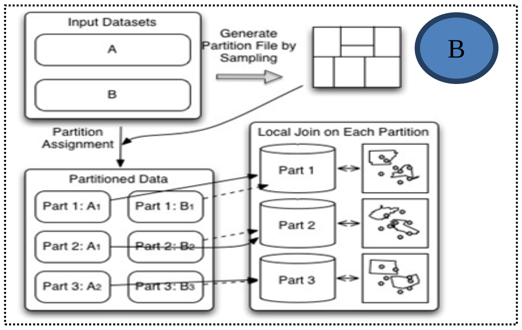
Clearly, when neighboring points are paired up with the same polygon, which is the case for many real world applications (e.g., the majority of taxi pick up locations are at a limited number of hot spots in NYC and a large portion of species are distributed in Amazon rainforest), the spatial refinement performance is expected to be better on both multi-core CPUs (due to cache locality) and GPUs (due to coalesced memory accesses). As such, a simple optimization is to sort the candidate pairs based on polygon identifiers at the beginning of spatial refinement to ensure that points that are paired with the same polygon are processed by the same CPU thread or GPU thread block. The optimization has the same effect when indexing point data using either a Grid-file or a Quadtree as discussed in [3]. However, since sorting can be expensive, the effectiveness of the optimization heuristic needs to be carefully considered. While the efficiency of both spatial filtering and spatial refinement will inevitably affect end-to-end performance, as the focus of this study is on the effectiveness of the lightweight distributed execution engine, we leave the optimization for future work. In addition, while we currently use HDFS for both distributing large data and broadcasting small data to worker nodes, the execution engine can be modified to allow data distribution and broadcast through network interfaces. This might provide higher performance, especially when the input datasets in a spatial join are produced by previous steps and stored in memory, which is also left for our future work.



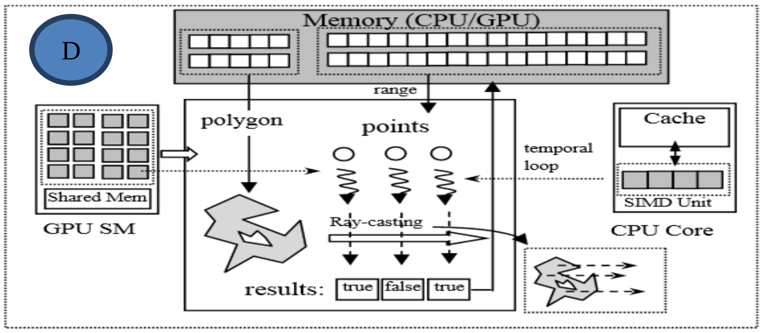
Broadcast based Spatial Join



Data Parallel Batched R-Tree Query for Spatial Filtering



Spatial Partition based Spatial Join



Data Parallel Point-in-polygon based spatial Refinement

Fig. 2 Illustration of Parallel and Distributed Spatial Join Techniques

IV. Experiments and Performance Evaluation

A. Setup

We use real world datasets to demonstrate the feasibility and efficiency of the proposed distributed point-in-polygon test based spatial join technique on top of the lightweight distributed execution engine. For point data, we use the geometry of a subset of species occurrence records of the Global Biodiversity Information Facility (GBIF). As the number of points is close to 50 million, we term it as *g50m*. The total volume of point data is 953MB. The polygon dataset is the World Wild Fund (WWF) global ecological region data[22] (termed as *wwf*). The number of polygons is 14,458 and the average number of polygon vertices is 279. The total polygon data volume is 149.8 MB. The same dataset has been used in ISP-MC+ and ISP-GPU on Amazon EC2 clusters [6]. It is thus interesting to compare the performance of the LDE engine on both multi-core CPUs (termed as LDE-MC) and GPUs (termed as LDE-GPU) with ISP-MC+ and ISP-GPU, respectively. We note that being able to store non-relational data (including geometry) and their indices in binary format in HDFS has reduced the data volume by several times in LDE than in ISP (as restricted by Impala), which is an important contributing factor to the efficiency of LDE.

Our experiments are performed on both a high-end GPU-equipped workstation and a 10-node Amazon EC2 *g2.2xlarge* GPU cluster[23]. The workstation is equipped with dual 8-core Intel Sandy Bridge 2.6 GHZ CPUs, 128 GB memory, 8 TB HDD and an NVIDIA GTX TITAN GPU. The GTX TITAN GPU has 6 GB GDDR5 memory and 2,668 CUDA cores. All Amazon *g2.2xlarge* instances (virtualized computing nodes) are equipped with 8 vCPU (Intel Sandy Bridge 2.6 GHZ), 15 GB memory, 60 GB SSD and an NVIDIA GPU with 4 GB graphics memory and 1,536 CUDA cores. All machines are running CentOS 6.5 and Hadoop 2.5.0 from Cloudera CDH 5.2.0 with default settings.

We design two groups of experiments to test the efficiency and scalability of our LDE engine. First, we experiment on the single-node performance and system infrastructure overhead (incurred by the LDE engine) on the workstation by comparing with a native implementation using the same spatial join designs. Second, we experiment on the scalability of LDE-GPU and LDE-MC by using 2-10 Amazon EC2 instances.

B. Results of *single*-node Performance

The standalone performance and the single-node performance for the two experiments are listed in Table 1. Note that ISP and LDE has the same runtime when they run in the standalone mode, which is 350 seconds on multi-core CPUs and 174 seconds on GPUs on the workstation. The runtimes in the single-node mode, however, are different among the four versions, which are 380 seconds for ISP on multi-core CPUs (ISP-MC+), 377 seconds for LDE on multi-core CPUs (LDE-MC), 241 seconds for ISP on GPUs (ISP-GPU) and 221 seconds for LDE on GPUs (LDE-GPU). It is clear that the GPU implementation performs about 2X (350/174) faster than the multi-core CPU implementation in the standalone mode. However, the infrastructure overhead has reduced the speedup to 1.58X (380/241) for ISP and 1.71X (377/221) for LDE. Nevertheless, by comparing Column 3 and Column 4 of Table 1 we can see that LDE has lower infrastructure overheads than ISP on both multi-core CPUs (27s vs. 30s) and GPUs (47s vs. 67s). The 20 seconds difference between LDE and ISP on GPUs have brought the infrastructure overhead from 27.80% (for ISP-GPU) to 21.27% (for LDE-GPU), which clearly demonstrates the efficiency of LDE design and implementations. It is also interesting to observe that the GPU implementations have higher percentages of infrastructure overheads than the CPU implementations. This is primarily because the floating point computing portion of the experiment has been significantly sped up by GPU while the speedup is not as significant as those on multi-core CPUs.

Table 1 Performance Comparisons between ISP and LDE in Standalone and Single-Node Modes

		Standalone Time (s) [A]	Singe-node Time (s) [B]	Infrastructure Overhead (%) (1-A/B)
CPU	ISP-MC+	350	380	7.89%
	LDE-MC		377	7.16%
GPU	ISP-GPU	174	241	27.80%
	LDE-GPU		221	21.27%

C. Results of Cloud Performance

The scalability results using 2-10 Amazon EC2 nodes are plotted in Fig. 3. We have avoided reporting the performance on a single node as it requires at least two nodes to count network communication overheads.

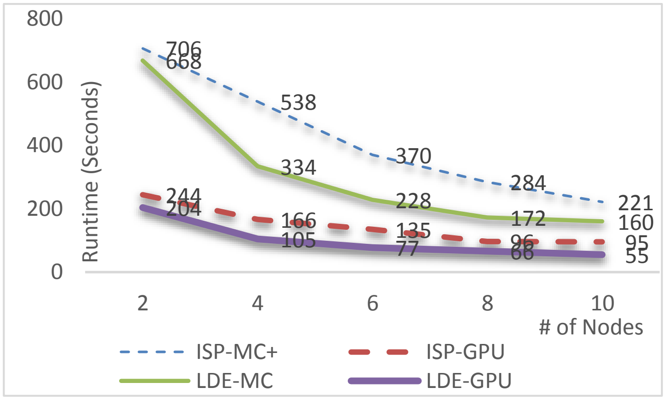


Fig. 3 Scalability Comparisons between ISP and LDE on Multi-core CPU and GPU Equipped Clusters

As shown in Fig. 3, when the number of nodes is increased from 2 to 10 (5X), the runtime is sped up 4.17X on multi-core CPUs (668/160) and 3.71X on GPUs (205/55). The speedups are higher than those in the ISP implementations, which are 3.19X (706/221) for multi-core CPUs and 2.56X for GPUs (166/95). The LDE implementations also have achieved significantly higher efficiency than the ISP implementations, ranging from 1.06X to 1.65X for multi-core CPUs and 1.20X to 1.75X for GPUs. Using 10 nodes, LDE is 1.38X faster than ISP for multicore CPUs (221/160) and 1.72X faster for GPUs (160/55). While the runtime using 10 nodes virtually remains the same as using 8 nodes for ISP on GPUs (1.25X increase of nodes), LDE is able to further achieve 1.20X (66/50) speedup, which is impressive.

V. Conclusion and Future Work

Motivated by lack of support for processing large-scale spatial and other types of non-relational data efficiently on mainstream Big Data systems, we have developed a lightweight distributed execution engine for spatial join query processing. The engine, built on top of the open source Apache Thrift and Boost libraries, is lightweight, friendly to indexing and native parallelization tools, and has high performance. Experiments using a real world application that spatially joins approximately 50 million species occurrence records and 15 thousand complex polygons with millions of vertices have demonstrated both efficiency and scalability on both a workstation and Amazon EC2 clusters with 2-10 nodes.

For future work, we would like to extend our conceptual design and prototypical implementation to support more spatial operations, including spatial joins based on distances and nearest neighbor searching, and other types of semi-structured data. Second, we would like to investigate the tradeoffs between HDFS based and direct network communication based distributed data accesses with respect to system complexity and efficiency to guide our future improvements of the LDE engine. Finally, while we acknowledge the necessities of complexities of Big Data systems for functionality and robustness, we are in the process of investigating the possibility in adopting a compilation driven approach to integrating our LDE engine with domain specific data abstractions, indexing, and query processing for spatial data in a way similar to Dandelion [23] for non-spatial data. By generating application specific code that is capable of distributed data communication and execution, the small, hardware- and application-aware and semantically optimized code can potentially be more efficient than existing heavyweight Big Data systems.

ACKNOWLEDGEMENT

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

References

[1] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann Publishers Inc., 2005.

[2] E. H. Jacox and H. Samet, "Spatial Join Techniques," ACM Trans. Database Syst., vol. 32, no. 1, p. Article #7, 2007.

[3] J. Zhang, S. You and L. Gruenwald, "Parallel Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs," Information Systems, vol. 44, p. 134–154, 2014.

[4] M. Kornacker, A. Behm, et al. "Impala: A Modern, Open-Source SQL Engine For Hadoop," in Proc. CIDR'15.

[5] S. You, J. Zhang and L. Gruenwald, "Large-Scale Spatial Join Query Processing in Cloud," To appear in Proc. IEEE CloudDM'15.

[6] S. You, J. Zhang and L. Gruenwald, "Scalable and Efficient Spatial Data Management on Multi-Core CPU and GPU Clusters: A Preliminary Implementation based on Impala," To appear in Proc. IEEE HardBD'15.

[7] L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th edition, Morgan Kaufmann, 2011.

[8] M. Zaharia, M. Chowdhury et al, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in Proc. NSDI'12.

[9] S. Alsubaiee, Y. Altowim, et al, "AsterixDB: A Scalable, Open Source BDMS," Proc. VLDB Endow., 7(14), 1905-1916, 2014.

[10] A. Alexandrov, R. Bergmann, et al, "The Stratosphere Platform for Big Data Analytics," The VLDB Journal, 23(6), 939--964, 2014.

[11] K.-H. Lee, Y.-J. Lee et al, "Parallel Data Processing with MapReduce: A Survey," SIGMOD Rec., 40(4), 11--20, 2011.

[12] C. Doulkeridis and K. Norvag, "A survey of large-scale analytical query processing in MapReduce," The VLDB Journal, 23(3), 355-280, 2014.

[13] S. Sakr, A. Liu and A. G. Fayoumi, "The Family of Mapreduce and Large-scale Data Processing Systems," ACM Comput. Surv., 46,(1), pp. 11:1--11:44, 2013.

[14] Y. Lu, J. Cheng, D. Yan and H. Wu, "Large-scale Distributed Graph Computing Systems: An Experimental Evaluation," Proc. VLDB Endow., vol. 8, no. 3, pp. 281--292, 2014.

[15] Y. Huai, A. Chauhan, et al, "Major Technical Advancements in Apache Hive," in Proc. ACM SIGMOD '14

[16] A. Aji, F. Wang, et al, "HadoopGIS: A High Performance Spatial Data Warehousing System over Mapreduce," Proc. VLDB Endow., vol. 6, no. 11, pp. 1009--1020, 2013.

[17] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in Proc. ICDE'15.

[18] K. Al-Naami, S. Seker and L. Khan, "GISQF: An Efficient Spatial Query Processing System," in IEEE CLOUD'14.

[19] Y. Gu and R. Grossman, "Toward Efficient and Simplified Distributed Data Intensive Computing," IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 6, pp. 974-984, 2011.

[20] J. Urbani, A. Margara, ro, C. Jacobs, S. Voulgaris and H. Bal, "AJIRA: A Lightweight Distributed Middleware for MapReduce and Stream Processing," in Proc. IEEE ICDCS'14.

[21] X. Lu, F. Liang, et al, "DataMPI: Extending MPI to Hadoop-Like Big Data Computing," in Proc. IEEE IPDPS'14.

[22] K. Ousterhout, P. Wendell, et al, "Sparrow: Distributed, Low Latency Scheduling," in Proc. ACM SOSP '13.

[23] C. J. Rossbach, Y. Yu, et al, "Dandelion: A Compiler and Runtime for Heterogeneous Systems," in Proc. ACM SOSP'13.

[24] S. J. Krieder, J. M. Wozniak et al, "Design and evaluation of the gemtc framework for GPU-enabled many-task computing," in Proc. HPDC'14.

[25] T. Li, X. Zhou et al, "ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table," in Proc. IEEE IPDPS'13.

[26] S. You, J. Zhang and L. Gruenwald, "Parallel spatial query processing on GPUs using R-trees," in Proc. ACM BigSpatial'13

[1] http://bustime.mta.info/wiki/Main/Technology
[2] http://www.gbif.org/
[3] http://hadoop.apache.org/docs/r1.2.1/streaming.html
[4] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[5] http://en.wikipedia.org/wiki/Java_virtual_machine
[6] http://blog.cloudera.com/blog/2014/09/new-benchmarks-for-sql-on-hadoop-impala-1-4-widens-the-performance-gap/
[7] http://trac.osgeo.org/geos
[8] http://en.wikipedia.org/wiki/Message_Passing_Interface
[9] http://en.wikipedia.org/wiki/InfiniBand
[10] SciDB: http://www.paradigm4.com/
[11] http://www.boost.org/doc/libs/1_57_0/doc/html/boost_asio.html
[12] http://akka.io/
[13] http://www.boost.org/
[14] https://thrift.apache.org/
[15] http://netty.io/
[16] http://en.wikipedia.org/wiki/Well-known_text
[17] https://hive.apache.org/
[18] http://en.wikipedia.org/wiki/CUDA
[19] https://thrust.github.io/
[20] http://openmp.org/wp/
[21] https://www.threadingbuildingblocks.org/
[22] https://www.worldwildlife.org/biomes
[23] http://aws.amazon.com/ec2/instance-types/