LARGE-SCALE SPATIAL DATA MANAGEMENT ON MODERN PARALLEL AND DISTRIBUTED PLATFORMS

by

SIMIN YOU

A dissertation submitted to the Graduate Faculty in Computer Science
in partial fulfillment of the requirements for the degree of Doctor of Philosophy,
The City University of New York
2015

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy.

Dr. Jianting Zhang

Date — Chair of Examining Committee

Dr. Robert Haralick

Date — Executive Officer

Dr. Zhigang Zhu

Dr. Spiros Bakiras

Dr. Jun Zhang

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

LARGE-SCALE SPATIAL DATA MANAGEMENT ON MODERN PARALLEL AND DISTRIBUTED PLATFORMS

by

SIMIN YOU

Adviser: Dr. Jianting Zhang

Rapidly growing volume of spatial data has made it desirable to develop efficient techniques for managing large-scale spatial data. Traditional spatial data management techniques cannot meet requirements of efficiency and scalability for large-scale spatial data processing. In this dissertation, we have developed new data-parallel designs for large-scale spatial data management that can better utilize modern inexpensive commodity parallel and distributed platforms, including multi-core CPUs, many-core GPUs and computer clusters, to achieve both efficiency and scalability. After introducing background on spatial data management and modern parallel and distributed systems, we present our parallel designs for spatial indexing and spatial join query processing on both multi-core CPUs and GPUs for high efficiency as well as their integrations with Big Data systems for better scalability. Experiment results using real world datasets demonstrate the effectiveness and efficiency of the proposed techniques on managing large-scale spatial data.

Contents

List of Tables

List of Figures

Chapter 1 Introduction

Recently, the fast growing data volume brings significant challenges on managing datasets at very large scale. It motives the development of emerging "Big Data" techniques for managing and analyzing the data. As most of information over the web includes spatial components, it is desirable to develop efficient techniques for large-scale spatial data, or "Big Spatial Data". For example, the increasingly available mobile devices have been generating tremendous amount of point data, such as locations collected using GPS. Advanced environmental observation and sensing technologies and scientific simulations have also generated large amounts of spatial data. For example, the Global Biodiversity Information Facility (GBIF[1]) has accumulated more than 400 million species occurrence records and many of them are associated with a location. It is essential to map the occurrence records to various ecological regions to understand the biodiversity patterns and make conservation plans.

On the other hand, parallel and distributed computing technologies have been developed to improve performance, including both hardware and software. The recent hardware developments include multi-core CPUs and emerging GPGPU (General Purpose computing on Graphics Processing Units) technologies. Also, memory capacity is getting larger, which motivates efficient in-memory processing techniques. On the software side, there are two major improvements over the recent decade. One improvement includes modern programming tools for multi-core CPUs and many-core GPUs, which make massive parallel computing power accessible for general public. The other improvement is the development of Big Data

[1] http://data.gbif.org

technologies, e.g., MapReduce [19] and its open source implementation Apache Hadoop[2], which allows using simple computing models to process large-scale datasets on distributed computing systems without deep knowledge in parallel and distributed computing. However, these platforms are primarily designed for relational data and may not be efficient or even suitable for spatial data.

Existing serial computing techniques for managing spatial data [82] usually focus on accelerating spatial data processing on single core CPUs, which are not suitable to process spatial data at very large scale especially when the data is beyond the capacity of a single machine. Although parallel techniques have been proposed for processing spatial data over the past few decades [82], most of them have not been able to take advantages of state-of-the-art parallel and distributed platforms. To alleviate the gap between the available computing power of parallel and distributed platforms and the practical needs on large-scale spatial data processing, we have developed techniques that can efficiently manage large-scale spatial data on modern parallel and distributed platforms. First of all, we have presented new parallel designs, including parallel spatial indexing and query processing techniques, for large-scale spatial data management. Second, we have investigated on how to implement such parallel designs using parallel primitives that are efficiently supported by many modern parallel platforms to achieve interoperability and productivity. Last but not least, we have developed relevant techniques to scale out spatial data processing to clusters that are increasingly available in Cloud Computing.

[2] http://hadoop.apache.org

The major contributions of this dissertation are as follows. First, we have identified practical challenges in large-scale spatial data management, especially in spatial indexing and spatial join processing. Second, we have developed parallel designs that are capable of taking advantages of state-of-the-art parallel and distributed platforms to address the practical needs of high performance computing for large-scale spatial data. Third, we have implemented prototype systems based our parallel designs to demonstrate the feasibility of the introduced designs. Finally, extensive experiments have been performed to demonstrate efficiency of the designs and implementations. Performance results of multiple reference implementations are discussed to understand the advantages and disadvantages of exploiting different modern parallel and distributed platforms in processing large-scale spatial data.

The rest of this dissertation is organized as follows. Chapter 2 introduces background and related work of this dissertation. Chapter 3 presents designs and implementations of parallel and distributed spatial indexing techniques. Chapter 4 provides designs and implementations of large-scale spatial join, which scale up on single-node parallel platforms and scale out on multi-node distributed platforms. Chapter 5 conducts extensive experiments for performance study on the implementations of the introduced designs. Finally, Chapter 6 concludes this dissertation and outlines potential future work.

Chapter 2 Background and Related Work

2.1 Modern Parallel and Distributed Platforms

The recent development of parallel computing technologies generally exploits two levels of parallel computing power. The first level is single-node parallelization that tightly couples multiple processors within a single machine, such as multi-core CPUs and GPGPUs, to deliver high computing power. The second level is multi-node parallelization that aggregates computing power from multiple loosely coupled machines in a distributed way. Figure 1 illustrates a typical architecture of modern parallel and distributed platforms that will be investigated in this dissertation.

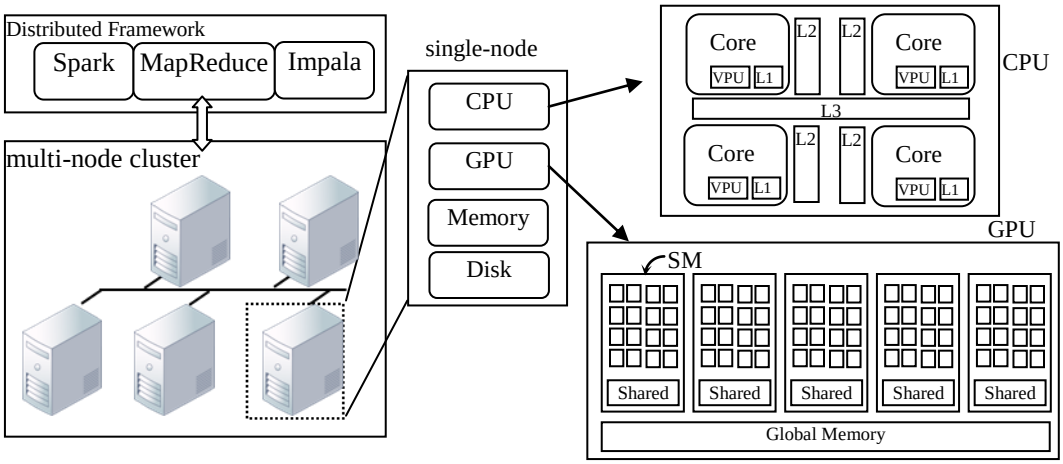


Figure 1 Parallel and Distributed Platforms

2.1.1 Single-Node Platforms

Parallel techniques have been developed on a single machine to deliver higher performance. An effort of increasing computing power on a single machine is to add more cores on a single CPU

socket (referred as multi-core CPU techniques), so that multiple tasks can be processed concurrently. Another effort is to use co-processors that are capable of providing massive parallel computing power, such as GPUs for general purpose computing (referred as many-core GPU techniques). All parallel processing units on the machines share the same memory space and they are considered as shared-memory systems.

2.1.1.1 Multi-core CPUs

While clock frequency on a single CPU core is nearly reaching physical limit, in the past few years, manufactures start to pack multiple cores into a single CPU socket in order to continue increase single CPU performance [38]. Today, almost every commodity computer has at least one multi-core CPU, which brings parallel computing to general public. Even for mobile phones, it is not uncommon to have a multi-core processor. However, there is still a significant gap between hardware and software as many software packages have not fully taken advantage of parallel hardware yet. To alleviate the gap, various parallel programming models have been developed. A common approach to utilize multi-core systems is using thread model, such as those based on OpenMP[3] and Intel Threading Building Blocks (TBB[4]) parallel libraries. In the thread model, computation is decomposed and distributed to all available cores in the form of software threads and all threads share the same memory space. This level of parallelism is termed as *task level parallelism*, where computation is divided into tasks and executed independently among threads.

[3] http://openmp.org

[4] http://threadingbuildingblocks.org

In addition to multi-cores, current CPUs usually have specialized hardware components such as *Vector Processing Unit* (VPU) to provide *Single-Instruction-Multiple-Data* (SIMD) capability[38]. With VPUs, each instruction can process multiple data items simultaneously. For instance, a 256-bit VPU can process eight 32-bit words in parallel. Thread level parallelism is then further enhanced by utilizing the specialized VPUs, which leads to another level of parallelism. Assuming there are *p* cores in a multi-core computing system and each core can perform SIMD operation on *v* items, the maximum number of parallel processing units in such a system is *p*v*. While most of existing works on parallel spatial data management only focus on utilizing available processing cores in parallel and distributed systems, it is possible to take advantage of VPUs which can further improve the overall performance. For relational data management, there are several works [54, 77, 109] successfully demonstrated the efficiency of utilizing SIMD operations. However, using SIMD computing power for spatial data processing is challenging for two reasons. First, SIMD instructions are usually restricted, and it is nontrivial to

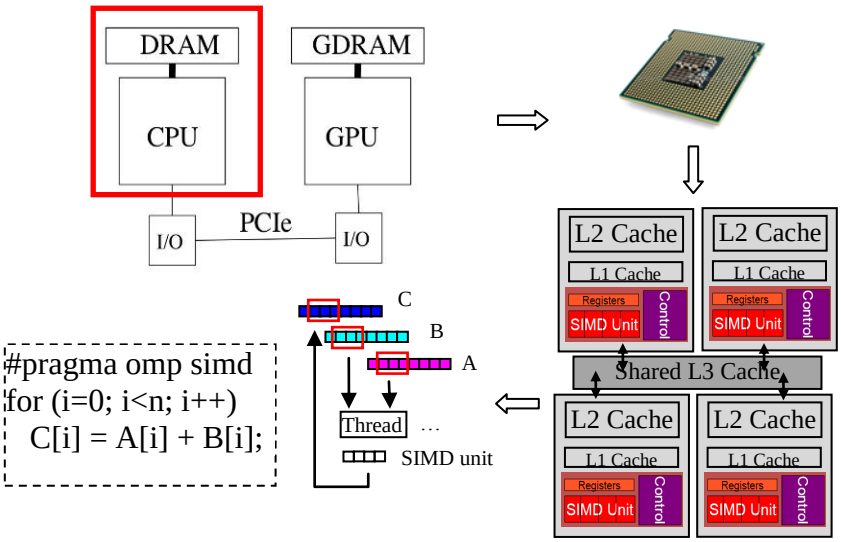


Figure 2 Multi-core CPU Architecture

identify which portions of spatial data processing are suitable for SIMD execution. Second, the memory access mechanism of SIMD units requires careful designs; otherwise it will result in low performance. Thus, memory access pattern in spatial data processing needs to be considered in order to achieve good performance.

Figure 2 shows an abstract architecture of multi-core CPUs including memory access hierarchy. Each core of the CPU has specialized SIMD units and private L1 and L2 caches, and there also exists shared L3 cache among CPU cores. The multi-level cache hierarchy aims to reduce expensive memory access time. The lower-left side of Figure 2 provides an example of adding up two arrays (A and B) and storing results to another array (C) using both threads and SIMD units. The workload is first divided into ranges, and each range is assigned to a thread for parallel processing. Then, within each thread, the range is further divided into batches which are processed by a SIMD unit in multiple rounds. Current CPUs also have limitations when used for large-scale spatial data management. First, memory access is expensive if memory hierarchy is not taken into consideration. When dealing with large-scale datasets, cache conscious data structures are critical for efficient memory access. For instance, dynamically allocated tree structures are very likely to result in significant cache misses during tree traversals. Second, irregular memory accesses can also result in serial executions on VPUs which is inefficient. Excessive use of memory gather/scatter operations might negatively impact SIMD performance as well. These challenges motivate us to develop data-parallel designs for large-scale spatial data processing that can be efficiently supported by current multi-core CPU platforms with SIMD capability.

2.1.1.2 GPGPUs

Traditional GPUs are dedicated accelerators for visual computing such as computer graphics, video decoding and 3D games. Unlike CPUs, GPUs have a large number of processing units which can perform computation on many pixels in parallel. Special function units (e.g. sine, cosine, reciprocal, square root) are also provided in GPUs to accelerate floating point computation in computer graphics applications. Many modern GPUs are capable of general computing and GPGPU technologies are becoming increasingly available, e.g., NVIDIA's Compute Unified Device Architecture (CUDA[5]) first appeared in 2007. Inheriting the advantage of using a large amount of processing units designed for graphical computing, GPGPUs can provide parallel computation by exploiting the general computing power of these parallel processing units. In this dissertation, we use GPU to refer to GPGPU unless otherwise explicitly stated.

A single GPU device consists of a chunk of GPU memory and multiple Streaming Multiprocessors (SMs). Each SM has multiple GPU cores; for example, there are 192 GPU cores on a SM and 14 SMs on an NVIDIA GTX Titan GPU. In the CUDA programming model, the parallel portions of an application executed on the GPU are called *kernels*. A kernel consists of multiple *computing blocks* and each block has multiple *threads.* During an execution, a computing block is mapped to a SM and each thread is executed on a GPU core. Notice that CUDA thread is different from CPU thread. A GPU core is typically weaker than a CPU core with lower clock frequency and much smaller caches. As a group of GPU cores (currently 32 in

[5] https://developer.nvidia.com/what-cuda

CUDA) in a computing block, called a *warp*, is only allowed to perform SIMD operations, GPU cores in a warp behave similarly to VPUs rather than CPU cores. All GPU cores within a warp can be considered as a VPU with a larger SIMD length (32*32=1024 bits). In addition, GPU cores assigned to the same computing block can use *shared memory* to share data. Different from CPUs that use large caches to hide memory latency, GPUs have much smaller caches but can use large numbers of computing blocks/warps to hide memory latency. Suppose the number of SMs on a GPU is *p* and each SM consists of *v* GPU cores, the total number of parallel processing units is then *p***v* which is similar to multi-core CPUs. However, *p***v* processing units on GPUs is significantly larger than that of multi-core CPUs. For instance, NVIDIA GTX Titan GPUs have 14 SMs and there are 192 GPU cores in a SM, which allows processing 14*192=2688 32-bit words simultaneously. In contrast, Intel X5405 CPUs only have 4 cores with 256-bit VPUs which can process 4*8=32 32-bit words in parallel.

Parallel computing on GPUs also has some disadvantages. The major problem is that

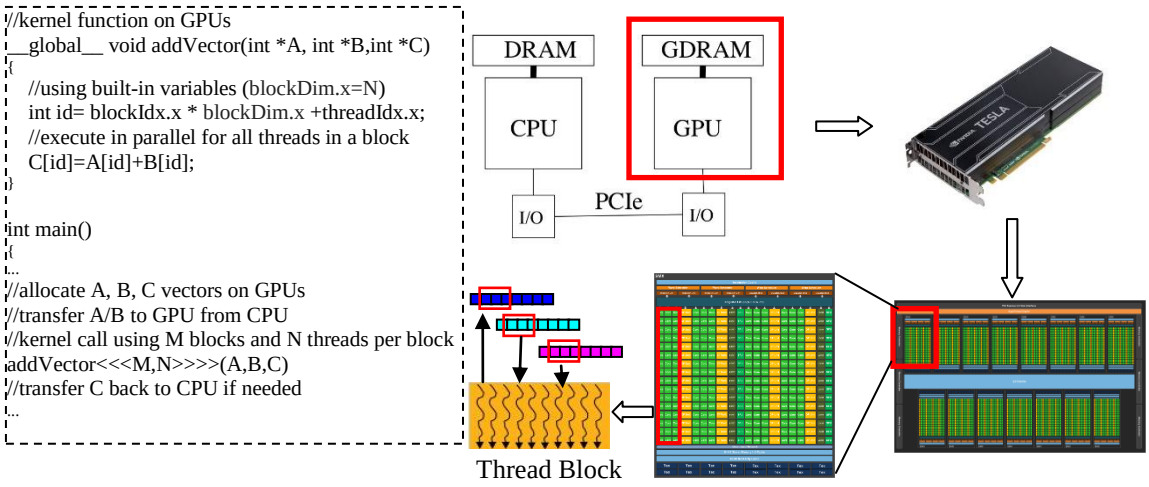


Figure 3 GPU Architecture and Programming Model

communication cost between CPU main memory and GPU memory is expensive. Currently GPUs are attached via PCI-E buses and data must be first transferred from CPU memory to GPU memory before performing computation on GPUs. Similarly, results need to be sent back to CPU memory for further processing after executions on GPUs. Because data transfer over a PCI-E bus is expensive (currently limited to 16GB/s for PCI-E 3 devices), the overall performance accelerated by GPUs might not be significant or even worse in some scenarios. In addition, GPUs typically have smaller memory capacity than CPUs, which can be a limiting factor in many applications. Even though GPUs can use pinned memory from CPU memory to virtually expand their memory capacities, the performance might be hurt due to data transfer overhead between CPU memory and GPU memory.

Figure 3 illustrates a typical GPU architecture and programming model. The left side of the figure shows an example of adding up two vectors in parallel on GPUs (using the CUDA model). The data is first transferred from CPU memory to GPU memory as shown in the first few lines of the main function. After that, the workload is divided into *M* blocks and each block uses *N* threads for computation. In CUDA, a block will be assigned to a physical SM for execution where each thread corresponds to a GPU core of the SM. Within a computing block, an index can be computed to address the relevant vector elements for inputs/outputs based on its thread identifier (*threadIdx.x*) and block identifier (*blockIdx.x*), which are automatically assigned by the hardware scheduler, and block dimension (*blockDim.x*), which is defined when the kernel is invoked.

GPU technology has been adopted in relational data management to accelerate database operators in the past few years [8, 27, 32, 36, 37, 93, 105]. Even before the existence of general purpose computing on GPUs, Bandi et al. [9, 10] has developed spatial selection and join query processing on GPUs using graphics rendering. As general purpose GPU computing becomes rapidly available in the past few years especially the development of CUDA programming model, many spatial data management techniques [7, 34, 49, 58, 65, 78, 79, 89, 90, 99–101], including spatial indexing and query processing, have been developed on GPUs.

2.1.2 Multi-Node Platforms

While many supercomputers in High-Performance Computing (HPC) centers have adopted distributed computing architectures and supported distributed computing over multiple computing nodes, they typically require users to adopt a pre-installed software stack such as Message Passing Interface (MPI[6]) libraries to simplify development and operation. Restricted accesses to HPC resources and steep learning curves on software tools have limited the adoptions of using HPC for Big Data applications. In contrast, Cloud Computing technologies have made it possible to rent cluster computers on-demand and pay-as-you-go with affordable prices for general public. New distributed computing tools, such as MapReduce [20] and its open source implementation Apache Hadoop[7], have made it much easier to develop and deploy parallel tasks on cluster computers provided by Cloud Computing vendors, such as Amazon EC2[8]. We next review two categories of distributed Big Data platforms, one is based on disk and the other

[6] http://www.mpi-forum.org

[7] https://hadoop.apache.org/

[8] http://aws.amazon.com/ec2/

further takes advantages of in-memory processing. Large-scale spatial data management on in-memory platforms can be significantly more performant than disk-based platforms, especially when GPU hardware accelerations are incorporated. On the other hand, disk-based platforms have longer history than in-memory platforms and are typically more robust and better supported. They may still be preferable when computing resources on individual computing nodes are limited.

2.1.2.1 Disk-based Platforms: MapReduce/Hadoop

MapReduce [20] is a parallel computing framework that is developed for processing large-scale datasets on large computer clusters. Unlike traditional cluster computing frameworks that require user to take care every aspect of parallel computing, MapReduce simplifies a parallel process into two steps, namely *map* and *reduce*. The *map* step divides input into sub-problems and sends them among all available nodes for distributed processing. The *reduce* step collects results from distributed nodes and assembles them into the final output. Users only need to write customized *map* and *reduce* functions and distributed execution is automatically accomplished by MapReduce runtime. Comparing with traditional parallel frameworks on clusters such as MPI, MapReduce is relatively simple and hides details of task scheduling and communication. A typical representation of MapReduce is as follows:

$$\begin{aligned} map\text{:} &\quad (key_1, value_1) \rightarrow \text{list}(key_2, value_2) \\ reduce\text{:} &\quad \big(key_2, \text{list}(value_2)\big) \rightarrow \text{list}(value_3) \end{aligned}$$

The user-defined *map* function converts the original problem into $(key_1, value_1)$ representation, and then the pairs are shuffled and distributed among all processing units automatically.

Subsequently each processor applies operations on $(key_1, value_1)$ in parallel and generates intermediate results, i.e., a list of $(key_2, value_2)$. Finally, the *reduce* function takes the intermediate results as input and reduces on key_2 to form the final output $value_3$ list.

A popular and widely used MapReduce implementation is Apache Hadoop. The Hadoop platform provides a dedicated distributed file system on top of operating system's file system, called Hadoop Distributed File System (HDFS). Data is stored in HDFS and is accessible to all computing nodes. MapReduce/Hadoop is a scalable system and has a relatively easy-to-use programming model. However, communication cost can be very high because data needs to be distributed to all computing nodes during the shuffling phase. For complex problems, decomposing the original problem using the MapReduce framework can be challenging due to the restrictive requirements of *map* and *reduce* operations. In order to utilize MapReduce, a problem may be decomposed in a suboptimal way that could potentially result in poor performance. The simplicity of MapReduce model brings scalability on large-scale data progressing; however, it may sacrifice expressive power and performance. Another issue of Hadoop based systems is that temporary results are written to HDFS, which sometimes can cause performance downgrade because of the excessive disk accesses which are very expensive.

2.1.2.2 In-memory based Platforms: Spark and Impala

As memory is getting significantly cheaper and computers are increasingly equipped with large memory capacities, there are considerable research and application interests in processing large-scale data in memory to reduce disk I/O bottlenecks and achieve better performance. Existing applications based on MapReduce/Hadoop have been praised for high scalability but criticized

for low efficiency [6]. Indeed, outputting intermediate results to disks, although advantageous for supporting fault-tolerance, incurs excessive disk I/Os which is getting significantly more expensive when compared with floating point computation on modern hardware and is considered a major performance bottleneck. In-memory big data systems designed for high performance, such as Apache Spark [106] and Cloudera Impala [14], have been gaining popularities since their inceptions.

From a user's perspective, Spark is designed as a development environment that provides data parallel APIs (Application Programming Interfaces) on collection/vector data structures, such as *sort*, *map*, *reduce* and *join*, in a way similar to parallel primitives. Spark is built on the notion of RDD (Resilient Distributed Dataset) [106] and implemented using Scala, a functional language that runs on Java Virtual Machines (JVMs). Compared with Java, programs written in Scala often utilize built-in data parallel functions for collections/vectors (such as *map*, *sort* and *reduce*), which makes the programs not only more concise but also parallelization friendly. Keys of collection data structures are used to partition collections and distribute them to multiple computing nodes to achieve salability. By using actor-oriented Akka communication module[9] for control-intensive communication and Netty[10] for data-intensive communication, Spark provides a high-performance and easy-to-use data communication library for distributed computing which is largely transparent to developers. Spark is designed to be compatible with the Hadoop ecosystem and can access data stored in HDFS directly. While Spark is designed to exploit large main memory capacities as much as possible to achieve high performance, it can spill data to

[9] http://akka.io/

[10] http://netty.io/

distributed disk storage which also helps to achieve fault tolerance. Although hardware failures are rare in small clusters [52], Spark provides fault tolerance through re-computing as RDDs keep track of data processing workflows. Recently, a Spark implementation of Daytona GraySort, i.e., sorting 100 TB of data with 1 trillion records, has achieved 3X more performance using 10X less computing nodes than Hadoop[11].

When comparing Spark with Hadoop, although both of them are intended as a development platform, Spark is more efficient with respect to avoiding excessive and unnecessary disk I/Os. MapReduce typically exploits coarse-gained task level parallelisms (in *map* and *reduce* tasks) which makes it friendly to adopt traditional serial implementations. Spark typically adopts parallel designs and implementations with fine-grained data parallelisms. The computing model adopted by Spark provides a richer set of parallel primitives not limited to *map* and *reduce* in MapReduce. The required efforts for re-designs and re-implementations of existing serial designs and implementations are very often well paid-off with higher performance, as programs expressed in parallel primitives based functional descriptions typically exhibit higher degrees of parallelisms and better optimization opportunities. With Spark, a problem represented by parallel primitives usually is less error-prone. A Spark cluster consists of a master node and multiple worker nodes. In runtime, the master node is responsible for coordination and dispatching workload to all worker nodes for execution.

Different from Spark, Impala is designed as an end-to-end system for efficiently processing SQL queries on relational data. It is an efficient Big Data query engine, which is

[11] https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html

considered as a replacement of Apache Hive[12] (compiles SQL statements to MapReduce jobs for execution) for interactive queries. In Impala, a SQL statement is first parsed by its frontend to generate a logical query plan. The logical query plan is then transformed into a physical execution plan after consulting HDFS and Hive metastore to retrieve metadata, such as the mapping between HDFS files and local files and table schemas. The physical execution plan is represented as an Abstract Syntax Tree (AST) where each node corresponds to an action, e.g., reading data from HDFS, evaluating a selection/projection/where clause or exchanging data among multiple distributed Impala instances. Multiple AST nodes can be grouped as a plan fragment with or without precedence constraints.

An Impala backend consists of a coordinator instance and multiple worker instances. One or multiple plan fragments in an execution plan can be executed in parallel in multiple work instances within an execution stage. Raw or intermediate data are exchanged between stages among multiple instances based on the predefined execution plan. When a set of tuples (i.e., a row-batch) is processed on a data exchange AST node, the tuples are either broadcast to all Impala work instances or sent to a specific work instance using a predefined hash function to map between the keys of the tuples and their destination Impala instances. Tuples are sent, received and processed in row batches and thus they are buffered at the either sender side, receiver side or both. While adopting a dynamic scheduling algorithm might provide better efficiency, currently Impala makes the execution plan at the frontend and executes the plan at the backend. No changes on the plan are made after the plan starts to execute at the backend. This

[12] https://hive.apache.org

significantly reduces communication complexities and overheads between the frontend and the backend which could make Impala more scalable, at the cost of possible performance lose.

As an in-memory system that is designed for high performance, the raw data and the intermediate data that are necessary for query processing are stored in memory, although it is technically possible to offload the data to disks to lower memory pressure and to support fault tolerance. An advantage of in-memory data storage in Impala is that, instead of using multiple copies of data in *map*, *shuffle* and *reduce* phases in Hadoop, it is sufficient to store pointers to the raw data in intermediate results, which can be advantageous than MapReduce/Hadoop in many cases, especially when values in (key, value) pairs have a large memory footprint.

2.2 Spatial Indexing Techniques

Spatial indexes are used by spatial databases to accelerate spatial queries. Various types of spatial indexes have been developed in the past few decades to support efficient spatial data access in many scenarios [30, 82]. In this section, we briefly introduce three major spatial indexes that are related to this research, i.e., Grid-files [70, 82], Quadtrees [28, 83] and R-trees [12, 35, 85, 103]. The major characteristics of the three categories of spatial indexes are tabulated in Table 1. The details will be discussed in the following subsections. As a common practice, for complex spatial objects such as polylines and polygons, instead of indexing on the exact geometry of spatial objects, Minimum Bounding Rectangles (MBRs) are used to approximate the geometry of spatial objects. As illustrated in Figure 4, MBRs are axis-aligned rectangles and can be efficiently derived from original objects.

Table 1 Summary of Spatial Indexes

	Grid-file	Quadtree	R-tree
Partition Strategy	space-oriented	space-oriented	data-oriented
Hierarchical Structure	No	Yes	Yes
Parallelization friendly	Good	Medium	Poor
Skewness Handling	Poor	Medium	Good

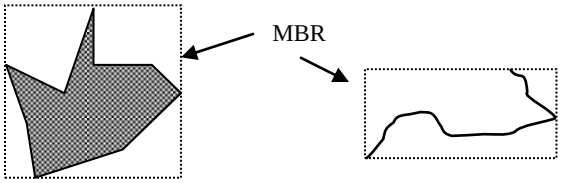


Figure 4 MBR Examples

2.2.1 Grid-Files

Grid-file [70] is a simple spatial data structure developed for efficient spatial data access and an example is shown in Figure 5a. To build a grid-file index, two parameters need to be specified first. One parameter is the extent of the indexing space which can be derived by scanning the input dataset being indexed. The other parameter is the size of grid cell, which is chosen empirically. After the initial parameter setup, MBRs are extracted from the original spatial objects. The MBRs are then mapped to the grid space according to the size of grid cell. If a MBR is larger than a single grid cell, it will be duplicated in all intersected grid cells. For example, object *A* in Figure 5a is duplicated in four grid cells (i.e., 1, 2, 5, 6). For range queries, the query processing is almost identical to index construction where the query window is mapped to the

same grid space and all intersected MBRs are retrieved using the matched grid cells. Since MBRs may be duplicated in the index construction phase, an additional duplication removal phase is required.

Based on how the space is decomposed, a grid-file can be categorized into non-uniform and uniform. For a non-uniform grid-file, the splitting points for each dimension are not uniformly distributed; so the splitting points need to be stored in order to locate each grid cell correctly. On the contrary, a uniform grid-file does not need to keep such information because the splitting points are uniformly distributed on each dimension and they can be derived from the extent of the space and the size of grid cells. In our research, we prefer uniform grid-file for simplicity. We will use grid-file to refer to uniform grid-file hereafter.

Unlike tree based hierarchical structures such as Quadtree and R-tree, a grid-file uses a flat structure that simply splits the space into grid cells, where each grid cell is a subspace that contains overlapping objects. The flat structure of grid-file indexing makes it parallelization friendly, because each grid cell can be processed independently and no dependency and synchronization between grid cells which are usually inevitable in hierarchical structures. The simplicity of grid-file has demonstrated its efficiency on modern parallel hardware, comparing with tree based indexes [73, 87, 88]. One drawback of grid-file indexing is skewness handling, especially for the uniform grid-file indexing. Since grid cells are generated by equally splitting the space, the number of objects in each grid cell can be very different on skewed datasets. The skewness will degrade index pruning performance and also create uneven workload that leads load balance issue in parallel computing. One way to partially address such issue is to choose a

good resolution for the space. However, finer resolution will incur another object duplicate issue. The issue is that, if an object overlaps with multiple grid cells, such object will be assigned to all overlapping cells. As such, an additional duplicate removal step is required when using grid-file indexing. Meanwhile, larger duplication imposes higher memory pressure, which could be a potential problem for memory constraint systems. Therefore, a good resolution parameter can be crucial to overall performance. Previous works [45, 102] have shown that both index construction and query processing can be significantly improved by using grid-file indexing on GPUs. Both [45] and [102] optimized the ray-tracing application using grid-file index on GPUs. Unlike previous works that focus on visualization, we exploit the potentials of utilizing parallel grid-file indexing for spatial data management. We also develop data-parallel designs using parallel primitives for grid-file based indexing, especially for supporting spatial join processing (Section 2.3). Recent works [33, 34] have adopted the idea of utilizing grid-file on the GPU for managing trajectory data.

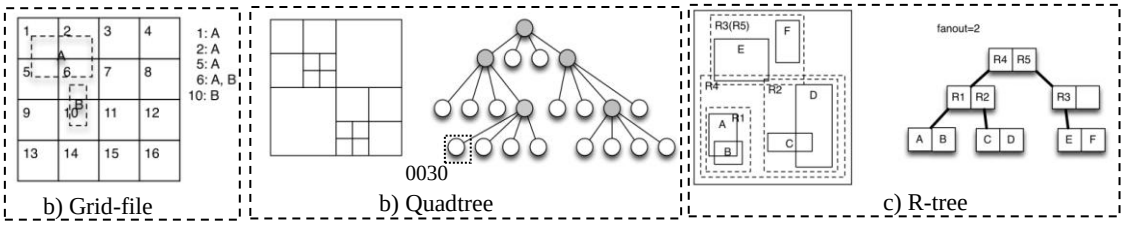


Figure 5 Spatial Index Examples

2.2.2 Quadtrees

Quadtree [28, 83] is a tree structure that is used for indexing spatial objects in 2-D space. It behaves similarly to binary trees in 1-D space. While there are many Quadtree variants, in this research, we use the term Quadtree to refer to Region Quadtree [82]. Region Quadtree follows

space-oriented decomposition and decomposes the whole space to be indexed into subspaces recursively until a certain criterion (e.g., minimum number of objects in the subspace, or the minimum size of the subspace) is met. Figure 5b illustrates an example of Region Quadtree, where each Quadtree node has four child nodes. Unlike R-tree to be introduced in the next subsection, Quadtree generates non-overlapping partitions that cover the whole space in a mutually exclusive and collectively exhaustive manner. Each node in the Quadtree, either leaf node or non-leaf node, is called a quadrant in Quadtree, which corresponds to a subspace. By the nature of Quadtree, each node is either decomposed into zero or four children. The four children are usually named NW (northwest), NE (northeast), SW (southwest) and SE (southeast) according to their relative locations. In a typical implementation of Quadtree on CPUs, each non-leaf node has four pointers pointing to its four children.

One feature of Quadtree is that each quadrant can be represented as a Morton code [82] which is a mapping based on Z-order [82]. The mapping can be realized by using 0, 1, 2, 3 to represent NW, NE, SW, SE nodes, respectively [31]. For example, the leftmost node in the last level of Figure 5b (enclosed in the dotted square) is represented as 0030. Such representation can be used to speed up range queries [2]. The regular splitting pattern of Quadtree is suitable for data-parallel designs. For example, the work in [39] took advantage of such feature to speed up spatial join processing. However, as Quadtree is a hierarchical data structure, there are dependencies between parent and child nodes. Comparing with grid-file indexing, it is technically challenging to develop a parallel Quadtree structure that can fully exploit parallelism. On the other hand, Quadtree splits the space using a threshold parameter that can alleviate the skewness issue as discussed in grid-file indexing. Even though dependency is an issue, it is still

attractive to use Quadtree indexing as a balance between parallelization and skewness handling. In this work, we will introduce data-parallel Quadtree construction and query algorithms on modern hardware such as multi-core CPUs and GPUs to support parallel spatial join processing.

2.2.3 R-trees

R-tree [35, 103] is a well known spatial indexing technique and has been widely adopted in many applications for indexing 2-D or higher dimensional spatial data. Similar to B-tree [18], an R-tree is also a balanced search tree but is adapted for multi-dimensional data. The key idea of R-tree is to group nearby objects and represent their aggregated spatial extent as a MBR. Unlike Quadtree that generates non-overlapping partitions, the spatial extents of R-tree nodes may overlap each other. On the other hand, R-tree typically follows data-oriented partition so that object duplication can be avoided. An example of R-tree is given in Figure 5c. In the example, we illustrate the R-tree with a fan-out of 2. The R-tree nodes are constructed from MBRs in the left of Figure 5c. For each entry in an R-tree node, a pair of MBR *M* and pointer *P* is stored, where the MBR *M* represents the union of all MBRs from its child node (e.g., R2 is the union of C and D) and the pointer *P* is used to access the child node corresponding to the entry.

An R-tree can be constructed via dynamic insertion or bulk loading. Dynamic insertion means the tree is constructed while MBRs are inserted one by one, which is suitable for indexing dynamic datasets. For static datasets, bulk loading might be more efficient. In bulk loading, an R-tree is constructed from the whole dataset typically by sorting and hierarchically aggregating MBRs bottom-up [74]. Querying on an R-tree is just like classic tree traversal, where MBRs stored at each node are used for spatial pruning. The query processing can be categorized into

two types, Depth-First-Search (DFS) based and Breadth-First-Search (BFS) based. To parallelize DFS based batch query, it is straightforward to assign each query to a parallel processing unit to query the tree individually. In such a design, each DFS query needs to maintain a small stack to keep track of intersected tree nodes. However, using DFS based query may incur load unbalance as queries usually follow different paths. The access pattern for DFS based query is also not cache friendly and not coalesced, which are important for parallel hardware such as GPUs. Previous work [63] suggested BFS based query processing can be more efficient on parallel hardware especially GPUs. Other works [50, 104] used a hybrid approach, in which R-tree was first traversed and then followed by a parallel linear scan.

In this work, we have improved parallel R-tree construction using parallel primitives (Section 3.2.3). The design is portable across multiple parallel platforms and improves the works reported in [63, 97]. We have also developed parallel primitive based designs for query processing which can serve as a module for efficient spatial join query processing.

2.2.4 Distributed Spatial Indexing Techniques

Most of the spatial indexing techniques developed in the past few decades focused on improving performance on a single computing node, and very few of them are developed for distributed environments [82]. Kamel and Faloutsos [47] proposed a parallel R-tree technique to support efficient range query. Observing that disk I/O was the dominating factor, they designed a parallel R-tree structure on a special hardware architecture which consisted of one CPU and multiple disks. In order to maximize throughput, R-tree nodes were distributed among all disks and linked

by cross-disk pointers. To answer a range query, R-tree nodes were loaded in parallel from disks and checked for intersection.

Koudas et al. [51] developed a parallel R-tree technique on a shared-nothing system. Instead of distributing R-tree nodes to multiple disks in [47], their design de-clustered R-tree nodes to multiple computing nodes. Another parallel R-tree structure on shared-nothing system is called *Master-client R-tree* proposed by Schnitzer and Leutenegger [84]. A master R-tree resided in a master node and its sub-trees called client trees were distributed on all client nodes. When a query arrived, it was processed on the master node sequentially and then distributed on client nodes to continue search in parallel.

Lai et al. [53] found that processing time on master node in [84] was a bottleneck and they proposed a different structure called *upgraded R-tree* which partitioned data space first and built an R-tree for each partition. By this means, the R-tree was distributed among all nodes and the bottleneck issue was solved. Mutenda and Kitsure [68] proposed a Replicated-Parallel-Packed R-tree (RPP-R-tree) technique which tried to minimize communication cost. The idea was to replicate R-tree among all nodes (by assuming disk storage cost was negligible). The master node was dedicated for task assignment and workload balancing. They developed a parallel spatial join approach using the proposed RPP-R-tree technique and claimed that their RPP-R-tree was more efficient for static data compared with dynamic R-tree used in [16].

The recent trend of distributed processing technologies, such as rapid development of Big Data platforms, motivates new designs and implementations of distributed spatial indexing techniques. Unlike earlier works introduced previously, the state-of-the-art distributed spatial

indexing techniques are designed for specific Big Data platforms (e.g., Hadoop). As those platforms usually provide restrictive data access models, most recently developed distributed spatial indexing techniques [3, 24, 98, 107, 108] are based on data repartition. In other words, spatial data are reorganized by spatial partitions where each partition contains a subset of the dataset. By performing a partition step, a spatial dataset is divided into a collection of independent subsets which can minimize unnecessary disk access and inter-node data transfer when processing a query. The spatial locality is preserved by spatially storing nearby data within a partition, which can accelerate related spatial queries such as nearest neighbor query.

VegaGiStore [108] was developed using the MapReduce model and running on top of Hadoop. In VegaGiStore, a global Quadtree based partitioning and indexing technique was provided, where each partition was represented as a quadrant of a global Quadtree and stored as a separate file with the calculated Morton code. Within each partition, a local index was saved as a file header and the rest of the file were spatial objects sorted according to Hilbert curve.

Hadoop-GIS [96] provided several partition strategies that can spatially partition data into tiles. In their work, spatial partitioning techniques were developed to solve the data skewness problem, which can significantly improve spatial query performance with MapReduce. An efficient and scalable partitioning framework named SATO [96] based on Hadoop was proposed, and the framework was implemented in four main steps: Sample, Analyze, Tear and Optimize.

Distributed spatial indexing is also supported in SpatialHadoop [24]. In the storage layer of SpatialHadoop, a two-level (including global and local) index structure was employed, which is similar to the idea of VegaGiStore. SpatialHadoop supported multiple spatial indexing

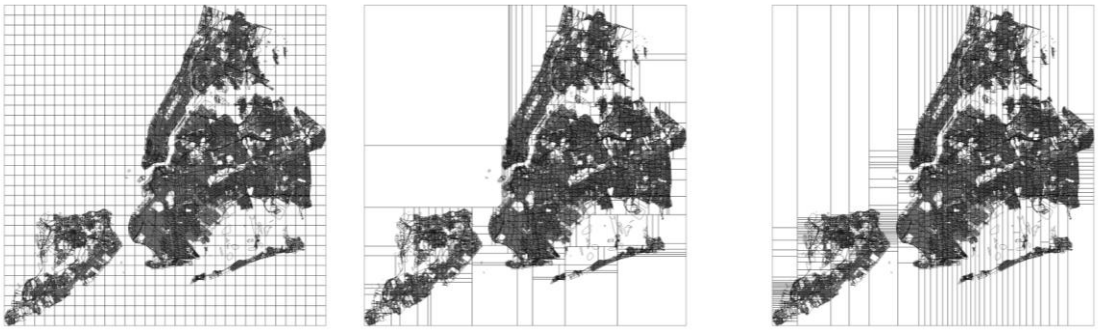
structures such as Grid-file, R-tree and R+-tree. The size of each partition was determined by HDFS block size, so that SpatialHadoop can achieve optimized disk access. The global indexing structure was physically stored as a master file on disk and can be loaded into memory while performing spatial query processing. Within each partition, SpatialHadoop stored a bulk-loaded local index at the beginning and it will be loaded while processing the particular partition.

GISQF [69] is an extension of SpatialHadoop that is developed to manage geo-referenced event database. *MD*-HBase [71] is a location based data management system on top of a key-value store, i.e., Apache HBase[13]. In their work, an additional multi-dimensional index layer was built for efficient data retrieval. Spatial objects (points) were encoded as bit strings according to Z-order, and queries were formalized as prefix matching. Li et al. [57] have developed Pyro, which is a spatial-temporal big data storage system also on top of HBase. However, different from *MD*-HBase, Pyro integrated spatial range query capacity into HBase system rather than making it an additional layer. Pyro also developed group based block replica placement that can preserve spatial locality for data storage. The shortcoming of both *MD*-HBase and Pyro is that they were developed for points rather than complex geometry objects, e.g., polygons. Since both systems relied on linearization, such as Z-order in *MD*-HBase and Moore encoding in Pyro [17, 95], it can be more challenging to extend them for complex geometries. Van and Takasu [94] recently developed an R-tree based distributed spatial indexing technique also on top of HBase. In their work, they designed a distributed spatial index structure using a combination of

[13] http://hbase.apache.org/

Geohash[14] and R-tree. Fox et al. [29] developed distributed indexing for NoSQL database, i.e., Apache Accumulo[15], where key-value store based design was adopted.

Since most of the state-of-the-art distributed spatial indexing techniques rely on spatial partitioning, partition quality will directly impact the performance of distributed processing. We will review three partition strategies, i.e., Fixed-Grid Partition (FGP), Binary Split Partition (BSP) and Sort-Tile Partition (STP) [96], which are related to this work. Those techniques are also integrated in our partition based spatial join in Section 4.2.1. Examples are provided in Figure 6 to illustrate the three spatial partition techniques, respectively.



Fixed-Grid Binary-Split Sort-Tile

Figure 6 Partition Examples

Fixed-Grid Partition (FGP) is the most straightforward way of space decomposition, where the whole space is divided into grid partitions with an equal size. This technique has been proposed and used in PBSM [76]. The choice of grid partition size heavily impacts the efficiency of FGP. When a large grid partition is chosen, fewer partitions will be generated. Using fewer partitions degrades the level of parallelism and also makes it difficult to process skewed data. To

[14] http://geohash.org

[15] https://accumulo.apache.org/

increase parallelism and handle data skewness effectively, one solution is to use finer grid partitions. With the improvement, more grid partitions are generated which is able to provide higher level of parallelism. Also, the straggler effect will be reduced if finer grid partitions are adopted. However, if an object crosses the boundary of multiple grid partitions, the object needs to be duplicated in each overlapping partition to ensure correctness. A finer grid partition will generate a larger number of duplications, which requires more memory during runtime. To sum up, FGP replies on the choice of grid partition, which typically impacts the overall performance as a "U" curve. To determine a good grid size, one solution is to perform selectivity estimation, and develop a cost model considering both data skewness and object duplication. Alternative solutions that can tackle skewness, such as using adaptive grid partition or multilevel grid partition (instead of using fixed-grid partition) can also be considered.

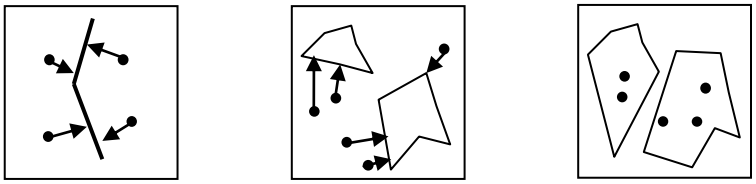
Binary Split Partition (BSP) is a partition strategy aims to produce balanced partitions, and partition boundaries are determined by data distribution rather than fixed in FGP. BSP first samples input data before splitting space into two subspaces and the process is done recursively. The splitting phase is very similar to the construction of K-D tree [13]. During an iteration step, a splitting dimension is chosen to split the space on the median point of the chosen dimension. The same procedure is recursively applied to the partitioned subspaces until the desired criterion is reached. The choice of splitting dimension can be based on the distribution of data as suggested in [96]. Meanwhile, a parameter defines the maximum number of recursive level, which controls the number of resulting partitions, needs to be introduced. In practice, constructing BSP from a large dataset can be time consuming. A single split needs a scan of the data for chosen dimension and a sort for calculating the splitting boundary. Even though single scan and sort could be

efficient on shared memory parallel platforms, multiple rounds of scan and sort operations require large amounts of data communication which may degrade performance in distributed computing environments. Besides, at each recursive step, the data will be reordered for the next iteration which also incurs significant data communication cost. One solution is to use a small portion of input dataset as a sample dataset to generate partitions on a single machine, if the sample is representative for the whole dataset. The BSP principle is also applicable to Quadtree based partition, which can be done by substituting the splitting phase with the Quadtree decomposition. More generally, the splitting phase can be replaced by any other recursive splitting approaches. Nevertheless, multiple rounds of scan and sort operations significantly lower the performance of BSP, which makes it less desirable for large datasets.

Sort-Tile Partition (STP) is proposed to generate partitions more efficiently. The technique is similar to the first step of Sort-Tile-Recursive R-tree (STR R-tree) bulk loading [55]. 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. STP can be adapted to strip-based partition by setting the number of splits on the secondary dimension to one, which essentially skips the second sort and split. Also, by first projecting data according to a space-filling curve (e.g, Z-order, Hilbert curve), using the same strip-based adaption can easily generate partitions based on the space-filling curve ordering. Different from BSP, STP at most sorts data twice and contains no recursive decompositions. Therefore, STP can be more efficient for large datasets.

2.3 Spatial Join Techniques

In a "Big Data" era, large-scale data analysis tools are highly demanded to analyze huge volume of spatial data that are generated every day. For example, with the fast growing smart phone market, tremendous amount of spatial data are generated from smart phones in the forms of GPS points and trajectories. To analyze the data, spatial join is required. For instance, answering a query such as "*find all smart phone users who are less than 100 meters to a movie theater*" needs a spatial join based on the "within distance" spatial relationship. However, it is not a trivial task to join huge amount of such data, especially when the spatial data is complex (e.g. polygon). In this section, we will first define the spatial join problem and then review existing works that have been developed to address the problem.



(a) Point to Nearest Polyline (b) Point to Nearest Polygon (c) Point in Polygon

Figure 7 Spatial Join Examples

2.3.1 Problem Definition

Spatial join can be formalized as follows. Given two spatial datasets R and S, the result of spatial join over R and S is,

$$\boldsymbol{R} \bowtie_{\boldsymbol{cond}} \boldsymbol{S} = \{(\boldsymbol{r},\boldsymbol{s}) | \boldsymbol{r} \in \boldsymbol{R}, \boldsymbol{s} \in \boldsymbol{S}, \boldsymbol{relation}(\boldsymbol{r},\boldsymbol{s}) \textbf{ is held}\},$$

where **relation** is a spatial relationship (usually a spatial predicate) between two spatial objects. Figure 7 gives three examples of spatial join based on point-to-nearest-polyline search, point-to-nearest-polygon search and point-in-polygon test, respectively. A naïve implementation of a spatial join is first to pair all objects from *R* and *S* and then to remove pairs that do not satisfy the spatial relationship in the spatial join. The naïve approach incurs a total complexity of $O(\|R\| \cdot \|S\|)$. However, spatial datasets are usually non-uniform and clustered and the naïve approach can be very inefficient. For example, in Figure 8, the naïve approach requires twelve intersection tests. However, if the space is indexed as partitions in advance and only objects in the same partition are paired, the number of intersection tests can be reduced to one. An intuition is that, if pairs can be pruned with little overhead before performing expensive geometric computation in testing spatial predicates, the overall performance can be improved. For this reason, *filter-and-refinement* strategy is adopted in most of existing spatial join techniques [43, 44].

The filter-and-refinement strategy divides spatial join processing into two phases, i.e., *filter* and *refinement*. In the filter phase, spatial objects are first approximated by axis aligned MBRs, and then stored in the form of $\langle OID, MBR \rangle$. Here OID is a pointer to the original spatial object and MBR refers to the extent of the spatial object. The approximated MBR representation

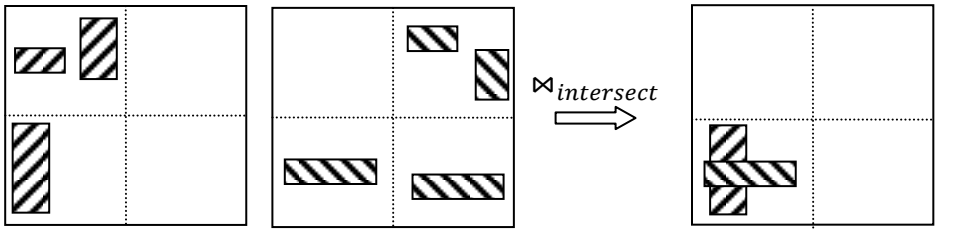


Figure 8 Intersection based Spatial Join Example

saves expensive geometric computation on the exact original spatial objects. For instance, the complexity of point-in-polygon test using the classic ray-casting algorithm is $O(n)$ where n is the number of vertices of the polygon being test. However, determining whether a point is in the MBR of a spatial object is only $O(1)$. Candidate pairs are generated and pruned with the MBR representation. Spatial access structures such as spatial indexes are usually used to reduce unnecessary candidate pairs and accelerate the pairing process. Denoting OID_r and OID_s as pointers to original spatial objects in *R* and *S*, the output of the filter phase can be represented as a list of $\langle OID_r, OID_s \rangle$.

For the filter phase, the most common spatial predicate on which prior works have studied extensively is *MBR intersection*, where two MBRs are checked on whether they spatially intersect each other. A running example of intersection based spatial join is given in Figure 8. Many other spatial relationship operators can be transformed into spatial intersection test. For example, the spatial join query operators such as *"within d"* and *"nearest neighbor within d"* can be realized by extending MBRs with distance d and subsequently performing spatial intersection join, as illustrated in Figure 9.

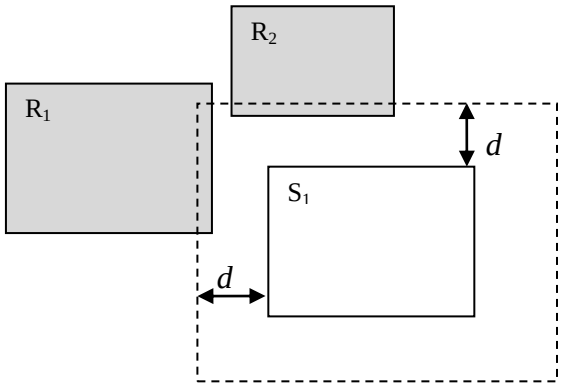


Figure 9 Spatial Join of WITHIN *d*

The filter phase prunes pairs that do not satisfy a spatial relationship but allows *false positives* because MBRs are used to approximate complex spatial objects. The refinement phase completely removes all false positives from the previous phase by testing the spatial relationship between two spatial objects based on their exact geometry. During the refinement phase, the exact geometric data are loaded using the OID_r and OID_s pointers. Spatial relationships are evaluated on the spatial objects by performing relevant geometric computation, such as point-in-polygon test. Due to expensive geometric computation as well as I/O costs of loading original objects, the false positive rate of the filter phase significantly impacts the overall performance of a spatial join. As such, most existing research has focused on optimizing the filter phase in order to minimize false positives.

Table 2 Summary of Spatial Join Techniques

	Plane-Sweep based	**Indexed Nested-loop based**	**Synchronized Index Traversal based**	**Partition based**
Pre-processing	Sort	Index construction	Index construction	Sort/Shuffle
Need Spatial Index?	No	Yes	Yes	No
Generate Duplicates in Output	No	Depends on index	Depends on index	Yes
Support Data-Parallel Design?	Very Difficult (Sequential in nature)	Easy	Difficult (Due to irregular data access on trees)	Moderate

For the rest of Section 2.3, we will discuss four leading spatial join techniques, including plane-sweep, indexed nested-loop, synchronized index traversal and partition-based. We will focus on parallelisms in discussing these spatial join techniques to set the context of this research on parallel spatial joins in Section 4.1 while refer to [44] for a more comprehensive survey on spatial joins. As a summary, Table 2 tabulates major characteristics of the four spatial joins that are relevant to our discussion. They will be detailed in the rest four subsections of Section 2.3.

2.3.2 Plane-Sweep based Spatial Join

The classic plane-sweep algorithm [86] reports all intersections from two sets of rectangles (MBRs in spatial joins) efficiently and has been widely used in spatial databases and Geographical Information System (GIS). The algorithm first sorts rectangles by their boundaries along one dimension (e.g., x axis). A vertical line then scans through the sorted list from left to right (or top to bottom). At any instant, a rectangle is considered active when it intersects with the sweep line. The key idea of this algorithm is, during the scan, a set of active rectangles are maintained and searched for reporting intersected pairs. To this end, the algorithm maintains a data structure, called *sweep structure*, to store active rectangles. Each time the sweep line meets a new rectangle, the sweep structure is updated where inactive rectangles are evicted and new active rectangles are inserted. Intersected pairs are then reported by searching on active rectangles. Various data structures, such as simple linked list, interval tries, interval tree and segment tree [44], have been adopted to support plane-sweep based spatial joins. Due to the initial sort before scan, the complexity of sweep plane implementations is at least $O(n \log(n))$, where n denotes the sum of the sizes of the two joining datasets. In the classic plane-sweep

algorithm, data are required to be loaded into memory first which restricts the scalability of plane-sweep based spatial joins.

To parallelize plane-sweep algorithm, a straightforward way is to divide the space to be swept into strips and apply plane-sweep algorithm on each strip. The parallelization of plane-sweep is known to be difficult as discuss in [66]. The authors attempted to implement parallel plane-sweep on multi-core systems by partitioning the space into multiple strips. In their design, pre- and post-processing steps were required to divide search space and merge results. During pre-processing, a scan was initiated to split input data into p strips. The p strips then ran plane-sweep algorithm individually in parallel. Finally, results were merged from all strips. There are several shortcomings in strip based parallel plane-sweep. First of all, choosing an optimal strip division is difficult. Using equal intervals on non-uniform distributed dataset usually results in unbalanced workload, which leads to poor performance. However, finding the optimal division is likely to impose more overhead of pre-processing, which might break the assumption that pre-processing overhead is negligible [66]. Second, parallel strip based approaches are limited in scalability. The parallelism for strip based plane-sweep is determined by the number of strips that can be processed in parallel. To maximize the performance, the number of strips needs to be at least equal or larger than the number of processing units. As the number of strips increases, post-processing overhead will also increase. From the analysis in [66], the complexity of post-processing is $O(pn \log pn)$ and it becomes inefficient when the number of strips (p) becomes large. Thus, strip based parallel plane-sweep is more suitable for processing small datasets or as a component in a larger framework (e.g. in [107]). Finally, the sequential scan in each strip restricts the granularity of parallelism, because such scan has dependencies which cannot be

broken down for finer granularity. Although there are many other parallel algorithms [11, 48, 59] have been developed for the plane-sweep problem, the intrinsic of plane-sweep algorithm limits its application in parallel spatial join domain. This characteristic is reflected in the last row of Table 2.

2.3.3 Indexed Nested-loop Spatial Join

Given two datasets *R* and *S*, if dataset *R* is indexed, indexed nested-loop join uses the other dataset *S* as a batch of queries on *R* to generate the join results. In the batch query, each element in *S* searches on the index of *R* with the desired spatial relationship and candidate pairs are reported if the relationship is met. For example, rectangle intersection based spatial join can be modeled as using one dataset as query windows to query the index of the other dataset. Given one dataset *R* with R-tree index and the other dataset *S*, and assuming the complexity for an element in *S* searching on the R-tree of *R* is $O(\log(n_r))$, then the complexity of indexed nested-loop join on R and S is $O(n_s \log(n_r) + \sigma)$ where σ is the additional overhead of generating intersection pairs. In many scenarios, spatial datasets have already been indexed using techniques such as R-trees and Quadtrees to boost spatial queries. Therefore, indexed nested-loop join can be realized relatively easily and no additional data structures are required. Figure 10 is the algorithm sketch of the indexed nested-loop join. Clearly, indexed nested-loop join is highly parallelizable (last row of Table 2) by assigning a data item in *S* to a processing unit and process all the items in parallel.

Luo et al. [63] implemented R-tree based batch query on GPUs. Their design supported a batch of window queries on an R-tree in parallel on GPUs, where each GPU computing block handled a single query in a Breadth-First-Search (BFS) manner [62]. The other approach of parallelizing indexed nested loop join is to use spatial indexes designed for parallelizing range queries. Kamel and Faloutsos [47] proposed parallel R-tree to support efficient range query. Observing that disk I/O was the dominating factor, they designed a parallel R-tree structure on a special hardware architecture which consisted of one CPU and multiple disks. To answer a range query, R-tree nodes were loaded in parallel from disks and checked for intersection. Koudas et al. [47] developed a parallel R-tree based on spatial join technique on a shared-nothing system. Instead of distributing R-tree nodes to multiple disks in [47], their design de-clustered R-tree nodes to multiple computer nodes. Another parallel R-tree structure on shared-nothing system called *Master-client R-tree* was proposed by Schnitzer and Leutenegger [84], where a master R-tree resided in a master node and its sub-trees called client trees were distributed on all client nodes. When a query arrived, the technique first processed it on the master node sequentially and then distributed it to client nodes to continue search in parallel. Lai et al. [53] found that processing time on the master node in [84] was a bottleneck and they proposed a different

```
Indexed_Nested_Loop_Join (R, S)
1. begin
2.   index_R ← Create_Index(R)
3.   foreach s ∈ S do
4.     results ← Index_Search(index_R, s)
5.     Report {⟨r, s⟩|r ∈ results }
6.   end
7. end
```

Figure 10 Indexed Nested-Loop Join Algorithm

structure called *upgraded R-tree* which partitioned data space first and built an R-tree for each partition individually. As a result, the whole R-tree was distributed among all nodes and the bottleneck issue was solved. In Hadoop-GIS [4], the authors also adopted R-tree based nested loop spatial join. The technique first partitioned the input datasets by sampling, and then, shuffled the datasets according to the generated partitions [96]. Each partition thus had a subset from both of the input datasets. Subsequently the indexed nested-loop join technique was applied within a partition while the whole spatial join can be parallelized at the partition level.

2.3.4 Synchronized Index Traversal based Spatial Join

When both datasets are indexed using tree based index, synchronized index traversal based spatial join can be used. Brinkhoff et al. [15] proposed using existing R-trees to speed up spatial joins by synchronized traversals from the roots of both R-trees, and nodes at same level were examined for spatial intersection. At each tree level during the traversal, a plane-sweep algorithm was used to report spatial intersections. Subsequently, intersected nodes were expanded and traversed until leaves were reached. If two trees did not have a same height, leaf nodes of the R-tree with lower height continued range queries on the rest sub-trees of the other R-tree. Huang et al. [42] optimized the original R-tree join in [15] using BFS traversal that achieved better performance; however, it had a drawback on controlling the priority queue size during the traversal.

Brinkhoff et al. [16] extended the sequential R-tree based join [15] to a shared-nothing parallel system. Similar to the sequential version, synchronized hierarchical traversal was used but sub-trees were sent to processors for parallel processing. On shared-nothing parallel systems,

in addition to CPU and I/O costs, network communication cost is also a crucial factor. A challenge identified in [15] was how to balance workload among processors during the execution with minimal communication overhead. Another parallel R-tree join technique on shared-nothing system was proposed by Mutenda and Kitsure [68]. They tried to minimize communication cost by proposing Replicated-Parallel-Packed R-tree (RPP-R-tree) as the spatial index. The idea was to replicate R-tree among all nodes (by assuming the disk storage cost was negligible). A master node was dedicated for task assignment and workload balancing. SpatialHadoop [24] implemented an R-tree based synchronized spatial join on Hadoop. When two datasets were indexed by R-tree, SpatialHadoop first generated the intersected partitions using a global R-tree. For each partition pair, synchronized spatial join was applied. For parallel R-tree joins on shared-memory systems, Yampaka and Chonstivatana [101] described a GPU based spatial join using R-tree. They used the same design from [16] but distributed the MBR intersection tests on GPUs instead of CPUs. During the spatial join, R-trees were bulk loaded before synchronized DFS traversals on the two R-trees. The traversals continued until leaf nodes were reached.

Besides R-trees, Quadtrees have also been adopted in parallel spatial joins. Hoel and Samet [39] developed a data parallel spatial join using PMR-Quadtree [82] on a hypercube machine. Starting from the root of two input datasets, nodes from the source and the target Quadtrees were matched and pairs were examined for spatial intersection in parallel. They demonstrated joining two polyline datasets based on the design. Hoel and Samet [39] also implemented R-tree based spatial join using the same hierarchical traversal design. Experiment study on both Quadtree and R-tree implementations showed that the Quadtree version outperformed the R-tree version significantly. The primary reason is that, on a data-parallel

computing platform, manipulating R-tree nodes, which are irregularly decomposed and are spatially non-disjoint, is more expensive than manipulating Quadtree nodes, which have non-overlapping spaces and a fixed number (i.e., four) of children.

As shown in the last row of Table 2, we rate the support for parallel designs in synchronized traversal based spatial join as “difficult”, mostly due to irregular data accesses on trees and the complexity in synchronizing the traversals on both trees.

2.3.5 Partition Based Spatial Join

Partition Based Spatial-Merge Join (PBSM) was proposed by Patel and Dewitt [76]. Similar to other spatial join algorithms, PBSM included the filter and refinement phases. However, PBSM did not build indexes if input datasets were not indexed. The data space was divided into partitions with a spatial partition function and each partition was assigned to a virtual processor to perform plane-sweep algorithm. If a MBR overlapped with multiple partitions, the MBR was duplicated and inserted into all overlapping partitions. Choosing a good spatial partition function was crucial for the performance. For example, as shown in the left side of Figure 11, partition 0

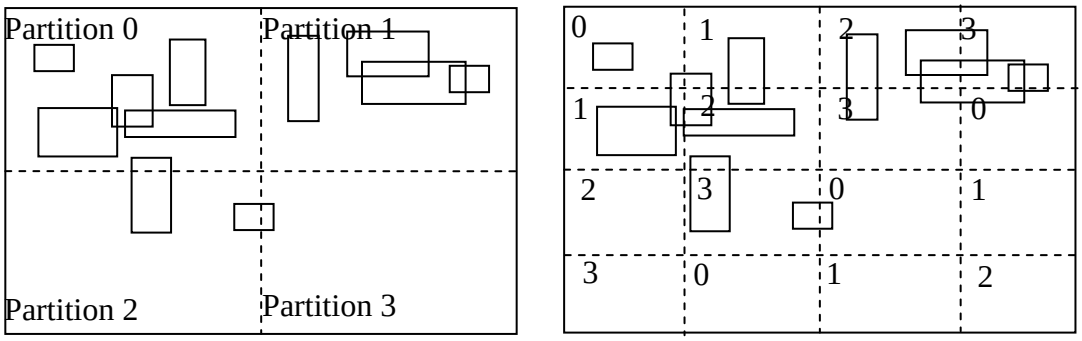


Figure 11 Tile-to-Partition and Skewed Spatial Data

and partition 1 are denser than other partitions. To address this issue, PBSM suggested a tile-to-partition mapping strategy. As illustrated in the right side of Figure 11, PBSM first divided space into tiles with finer granularity and then grouped them into coarser partitions to overcome unbalanced division. The space was first decomposed into $N * T$ tiles where $N * T$ was greater than P. Subsequently the tiles were assigned to partitions in a Round-Robin manner (or using hashing). After the filter phase, MBR pairs $\langle OID_r, OID_s \rangle$ were generated for the refinement phase. As duplicated MBRs were generated during partitioning, they could also be generated in the filter phase and needed to be removed. This could be realized by sorting or *Reference Point Method* (RPM) technique suggested in [22]. With RPM, duplicate pairs could be removed by checking whether the reference point fell within the partition without sorting which could be expensive.

Although the PBSM algorithm was developed for serial computing on a single CPU, the idea of using virtual processors can be naturally adapted to parallel computing. The implementation of Parallel PBSM (PPBSM) is straightforward by assigning each partition to a processor in a shared-nothing parallel environment. Patel and Dewitt [75] proposed two spatial join algorithms, *clone join* and *shadow join*, which are considered as improved versions of PPBSM. Clone join was identical to the spatial partition function used in the original PBSM, i.e., MBRs intersected with tiles were replicated and assigned to all intersecting tiles. Observing that there were large numbers of duplication generated in clone joins, finer object approximations were used in shadow joins in [75]. Instead of using a single MBR, a spatial object was approximated using multiple *fragment boxes*, where each fragment box was the MBR of the overlapped portion of the object and a tile. This design minimized the size of duplication by

creating partial surrogates. However, additional steps were required to eliminate partial surrogates to form candidate pairs for the refinement phase in shadow joins.

Niharika [80] is a parallel spatial data analysis infrastructure developed for Cloud environments which aims to exploit all available cores in a heterogeneous cluster. Niharika first uses a declustering technique that creates balanced spatial partitions and then dispatched workload to multiple workers. SPINOJA [81] is a system developed for in-memory parallel spatial join processing. In SPINOJA, a technique called MOD-Quadtree (Metric-based Object Decomposition Quadtree) is developed to handle skewness in order to produce better workload. Zhou et al. [111] have implemented PBSM on a dedicated parallel machine. They improved the original PBSM partition function by using Z-order curve [82] instead of the original Round-Robin assignment. The Z-order curve partition preserved better locality and achieved better performance according to their experiments. Zhang et al. [107] developed a variant of PPBSM called SJMR based on the MapReduce framework. SJMR adopted duplication avoidance technique named *reference tile method*, which considered checking whether the reference point fell within tiles rather than in partitions [22]. Zhong et al. [108] also implemented parallel spatial join on MapReduce platform using two-tier index which actually served as a partition function. To perform spatial join in the two-tier structure, overlapping partitions were matched and loaded through their filenames. In each partition, intersecting pairs were generated using an in-memory spatial join technique based on Hilbert Space Filling Curve [82]. Parallel SECONDO [61] introduced by Lu and Guting is another Hadoop-based solution which extends SECONDO [21] from a single machine to a Hadoop cluster.

In order to handle skewness, PBSM divides space into a large number of tiles. However, it is possible to group non-continuous tiles into a same partition (see the right side of Figure 11). Lo and Ravishankar [60] suggested the Spatial Hash Join (SHJ) technique to address this issue. Instead of decomposing space into regular gird tiles, SHJ generated buckets from one dataset, termed *inner* dataset. The other dataset, termed *outer* dataset, was overlaid on top of the inner buckets to pair MBRs from the outer dataset with the overlapping inner buckets. A recent technique called TOUCH [72] used an idea similar to SHJ. In TOUCH, an in-memory data structure similar to an R-tree was created from one dataset. MBRs from the other dataset were pushed down to the tree structure and assigned to different buckets. Unlike SHJ that retrieved all intersecting buckets for the query MBR, TOUCH found the minimal enclosing node and used all MBRs from the node as candidates. Even though larger false positives were generated, TOUCH avoided duplication and performed well due to contiguous memory access on modern hardware. THERMAL-JOIN [92] is another in-memory spatial join which is similar to TOUCH but yields better performance. The major improvement in THERMAL-JOIN comparing with TOUCH is the indexing structure. Instead of using tree structure in TOUCH, the new design adopted grid file based indexing, namely T-Grid and P-Grid, and it demonstrated significant speedup over TOUCH on dynamic workload. Partition based methods are also adopted by distributed systems such as Hadoop. Both Hadoop-GIS [3, 4, 96] and SpatialHadoop [23–26] adopted a two-step approach for distributed spatial join where the first step was dedicated to pairing spatial partitions. Different from Hadoop-GIS that used indexed nested loop in the second step within a partition as discussed in Section 2.3.3, SpatialHadoop also supported plane-sweep and synchronized index traversal.

With respect to supporting parallel designs, the parallelisms at the partition level in partition based spatial join are obvious and there are parallelization opportunities within partitions. Unlike indexed nested-loop spatial join where load balancing can be relatively easy achieved, it requires more efforts to avoid/remove duplicates and achieve load balancing in spatial partition based spatial join. For this reason, as indicated in the last row of Table 2, we rate the level of support for parallel designs in spatial partition based spatial join as “Medium”.

Chapter 3 Parallel and Distributed Spatial Indexing

3.1 Overview

We develop parallel designs of spatial data management at two levels. First, we would like to fully exploit the parallel computing power on a single computing node using commodity hardware such as multi-core CPUs and GPUs. We investigate on data structures and parallel algorithm designs for the new hardware, which can scale up spatial data processing on a single node. The second level is to scale out the single node parallel designs to multiple computing nodes, which provides scalable data management capabilities for larger scale spatial data. By achieving both efficiency and scalability, we expect our parallel and distributed techniques can significantly speed up processing large-scale spatial data using existing software packages, which are mostly designed for uniprocessors and disk-resident systems based on a serial computing model.

3.2 Parallel Spatial Indexing on Single-Node

In this section, we will introduce our designs on parallel spatial indexing on a single node. First, we will discuss our proposed spatial data layout that is efficient on both multi-core CPUs and GPUs. We will then introduce our parallel designs on three well-known spatial indexes, i.e., Grid-file, Quadtree and R-tree. While parallel designs of spatial indexes are mainly focused on single-node parallelization that utilizes multi-core CPUs and GPUs, they can be used as building blocks for distributed computing to be presented in Section 4.2.

3.2.1 Data Parallel Geometry Layout

Although several geometry representation formats such as Well-Known Text (WKT)[16] have been adopted in many existing software libraries, they were not designed for data-parallel operations and are not efficient on the current generation of parallel hardware, such as SIMD enabled processors. We have developed novel spatial data layout designs for efficient in-memory geometry operations, which are cache friendly and effective for data-parallel operations on both multi-core CPUs and GPUs.

Since Open Geospatial Consortium Simple Feature Specification (OGC SFS[17]) has been widely adopted by the Spatial Databases and GIS communities, our in-memory data structures for geometries are designed to support the standard. Taking polygon data as an example, according to the specification, a polygonal feature may have multiple rings and each ring consists of multiple vertices. As such, we can form a four level hierarchy from a dataset collection to vertices, i.e., dataset → feature → ring → vertex. In our design, five arrays are used for a large polygon collection. Besides the *x* and *y* coordinate arrays, three auxiliary arrays are used to maintain the position boundaries of the aforementioned hierarchy. Given a dataset ID (0..N-1), the starting position and the ending position of features in the dataset can be looked up in the feature index array. For a feature (polygon) within a dataset, the starting position and the ending position of rings in the feature can be looked up in the ring index array. Similarly, for a ring within a feature, the starting position and the ending position of vertices belong to the ring can be looked up in the vertex index array. Finally, the coordinates of the ring can be retrieved by accessing the *x* and *y* arrays. We note that for a single polygon dataset, the feature index array

[16] https://en.wikipedia.org/wiki/Well-known_text

[17] http://www.opengeospatial.org/

can be replaced by a constant to simplify the structure. Similarly, for polygons with a single ring, the ring index array can be skipped. Polyline datasets can follow similar designs where rings correspond to line segments. Point datasets can simply use the *x* and *y* arrays without the auxiliary arrays for polylines and polygons.

It is easy to observe that retrieving coordinates of single or a range of polygon datasets, features and rings can all be done by sequentially scanning the arrays in a cache friendly manner. It is also clear that the number of features in a dataset, the number of rings in a feature and the number of vertices in a ring can be easily calculated by subtracting two neighboring positions in the respective index array. As such, the array representation is also space efficient. Clearly, polygons using our proposed data layout are represented as Structure of Arrays (SoA) instead of Array of Structures (AoS), which is used in most of existing geometry representation including WKT. The use of SoA is potentially more efficient on modern parallel hardware because same data types are grouped together and exposed for better vectorization, especially on SIMD enabled devices such as VPUs and GPUs. Figure 12 gives an example of the SoA layout of a polygon collection. In the example, a polygon with identifier 50 stores ending positions (73, 78, …, 100) of its rings in the ring index array. Therefore, we are able to locate all rings belong to the polygon, which starts right after the last ring of the previous polygon (e.g., 70 in ring index array) and ends at the last ring (e.g., 100 in the ring index array). The ending vertex position of each ring is stored in the vertex index array. For example, the first ring of polygon 50 (73 in the ring index array) has an ending position of 913 in the example. By using the ending position from each vertex range, *x*/*y* coordinates are retrieved from the coordinate arrays.

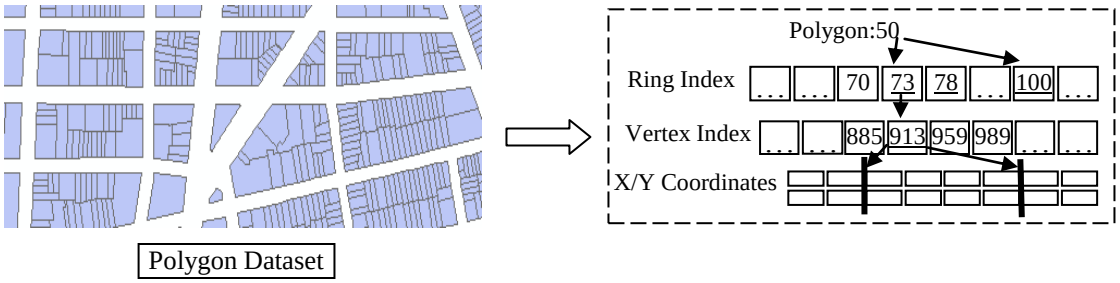


Figure 12 Spatial Data Layout Example

In addition to the exact representation of geometry objects, approximate representation such as MBR is also important because it is widely adopted in spatial indexing. We represent MBRs using four arrays to store the lower-*x*, lower-*y*, upper-*x* and upper-*y* coordinates, respectively. Extracting MBRs from the original geometry objects is embarrassingly parallelizable. The whole MBR extraction procedure can be easily implemented by using a single *reduce_by_key* parallel primitive (see Appendix A) with the vertex array as one input and the MBR id array as another input to specify keys. Figure 13 is an example of utilizing parallel primitives to extract MBRs from the spatial data layout we have developed. First, an auxiliary identifier array is allocated with the same length of the *x* or *y* array. The array is filled out by using a *scatter* and a *scan* primitive. The *scatter* primitive writes polygon identifiers to the newly allocated identifier array using the starting positions of polygon vertices. The partially filled identifier array is then completed with a *scan* primitive, which copies every identifier to its right until another identifier is met. The process is illustrated in the upper right of Figure 13. After generating the identifier array, a *reduce_by_key* is performed by using identifiers as the keys and coordinate arrays (*x* and *y*) are the reduction values. In *reduce_by_key*, the reduction values with

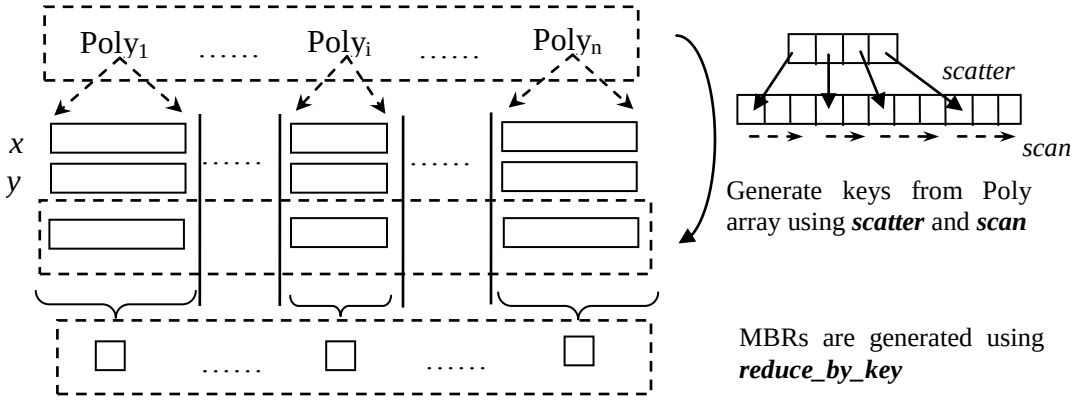


Figure 13 Extracting MBRs using Parallel Primitives

the same key will be applied with a pre-defined binary and associative operation, such as *min* or *max* function. Finally, the results are saved into the four arrays as introduced previously.

3.2.2 Parallel Grid-File based Indexing for MBRs

We first introduce an in-memory grid-file based spatial index on parallel platforms using data-parallel designs for MBRs of both polylines and polygons (Section 2.2). The designs are also applicable to points that can be considered as MBRs with a zero extent. The data-parallel grid-file index is designed to support efficient parallel spatial queries (this section) and spatial joins (Section 4.1). There are three major components in developing the parallel grid-file based indexing technique. First, we design the index layout using simple linear arrays that are efficient on both CPUs and GPUs as discussed previously. Uniform grid is chosen for simplicity and efficiency. Second, we develop a query strategy using binary search that is both efficient and requires no extra space. Third, for all the stages of index construction, our introduced data-parallel designs can be implemented using parallel primitives, which not only simplifies code complexity but also makes it portable across multiple parallel platforms.

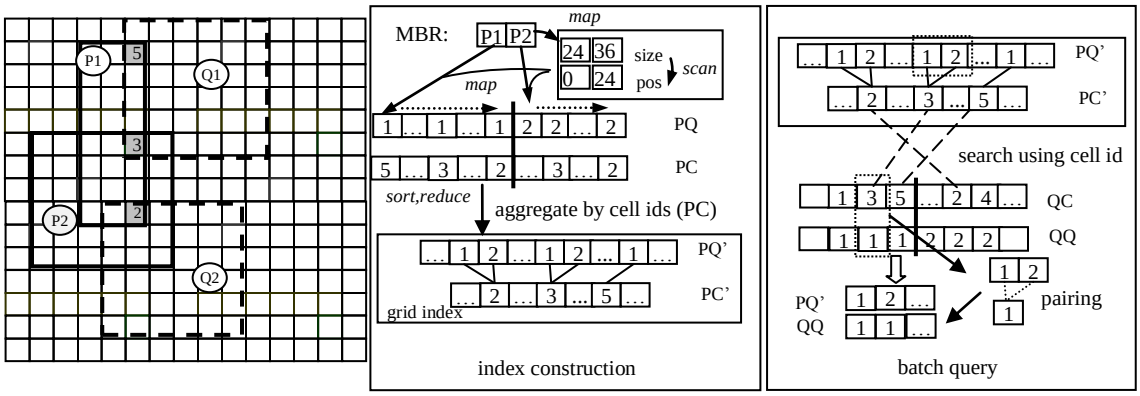


Figure 14 Parallel Grid-File based

The parallel grid-file indexing technique is based on the decomposition of a set of MBRs according to a uniform grid space whose resolution is chosen empirically. The grid-file index is constructed through projecting the input MBRs on the grid space followed by an aggregation operation. The projection is parallelizable by decomposing MBRs to grid cells in parallel by chaining a few parallel primitives, which will be illustrated using an example. The aggregation can be regarded as a parallel reduction operation where the grid cell ids are keys. We store grid-file index using simple arrays, including grid cell ids, MBR ids and an additional position array. The position array stores the ending positions of rasterized MBRs and links grid cell ids and MBR ids. In our design, only grid cells that intersect MBRs are stored for space efficiency.

The middle part of Figure 14 illustrates the procedure of constructing a grid-file index from two input MBRs. First, two MBRs (P1 and P2) are first projected to the grid space and the output sizes for the MBRs are calculated. A *scan* is performed on the output size array in order to compute the starting position of each MBR. With the starting positions and output sizes, each MBR is decomposed into cell id and MBR id pairs, which are stored in arrays PC and PQ,

respectively. Finally, the pairs are sorted by the cell id array. A *reduce_by_key* parallel primitive is applied to transform the cell id array from a dense representation into a sparse representation by keeping only the unique cell ids (PC') and the numbers of duplicated cells (PN) which represent the numbers of MBRs that intersect with cells. Note that, in the middle of Figure 14, array PQ' is the sorted copy of PQ; array PN, which keeps track of the connection between PC' and PQ', is skipped to simplify the illustration.

We also design parallel batch query processing using the grid-file based indexing, where a batch of range queries (i.e., window queries) are performed in parallel and intersected pairs are returned. Using the example shown in Figure 14, we assume that {P1, P2} and {Q1, Q2} are the indexed MBRs and query MBRs, respectively. Without using spatial index, the query needs to cross compare on all pairs, which is very expensive. To efficiently find the P-Q pairs that spatially intersect using the grid-file index, as illustrated in the right part of Figure 14, first, P1 and P2 are projected onto a grid space and indexed by arrays PC' and PQ' using the previously introduced procedure. Second, the query MBRs (i.e., Q1 and Q2) are projected to the same grid space and the results are stored in arrays QC and QQ. QC and QQ represent query MBRs and the order of the arrays will not affect the results. As such, for efficiency purpose, it is not necessary to sort and reduce QC or QQ to generate QC' and QQ'. Finally, the query is performed by matching the cell ids from the two sets of MBRs and the result pairs are generated based on matched cell ids. The details on matching are given next.

In classic designs based on serial computing, the matching process can be done by maintaining a hash-table for indexed MBRs with their cell ids. In contrast, our data-parallel

design chains several parallel primitives for the purpose. Since both the index MBRs and query MBRs are projected to the same space, we can link them using cell ids which can be implemented as a parallel binary search of all the elements in QC on PC'. For example, in the right part of in Figure 14, the query pair (3, 1) in QC and QQ array locates the corresponding cell in the index arrays (PC' and PQ'). Since the index arrays are sorted, the matching is done by performing a binary search on PC' for each query cell from QC. To speed up the process, we assign each query cell from QC to a thread so that the matching can be done by a parallel binary search. After the parallel binary search, each query cell is associated with a matched cell identifier from PC'. In the example, the query pair (3, 1) is matched with 3 in PC' and then identifiers 1 and 2 are retrieved from PQ' by using an auxiliary array that links PC' and PQ'. By performing parallel binary search on the sorted PC' array, each cell identifier from QC can be matched with a cell identifier in PC'. Then, identifiers from PQ' and QQ are further paired since they are directly connected with PC' and QC. As all the involved operations, i.e., *sort*, *search* and *unique*, can be efficiently parallelized in quite a few parallel libraries including Thrust[18] (that comes with CUDA SDK), batch spatial query using grid-file indexing can be relatively easily implemented on GPUs.

The process of grid-file based query processing transforms a spatial query problem (MBR intersection) into a non-spatial problem (binary search) that can be easily parallelized. However, the MBRs intersecting with multiple grid cells will be duplicated in each grid cell, which imposes additional memory pressure that can be a significant limiting factor on devices with

[18] https://thrust.github.io/

limited memory, such as GPUs. This issue can be partially addressed by tuning grid cell sizes. Clearly using larger grid cells will have smaller number of pairs but produce more false positives. Compared with the R-tree based spatial indexing to be introduced next, while parallel grid-file is simple in design and easy to implement, it typically requires larger memory footprint and should be used with caution.

3.2.3 Parallel R-tree based Indexing for MBRs

3.2.3.1 Data-Parallel R-tree Layout

Instead of using classic pointer based tree structure, we design simple linear array based data structures to represent an R-tree. As discussed previously, the simple linear data structures can be easily streamed between CPU main memory and GPU device memory without serialization/deserialization and are also cache friendly on both CPUs and GPUs. In our design, each non-leaf node is represented as a tuple {*MBR*, *pos*, *len*}, where *MBR* is the minimum

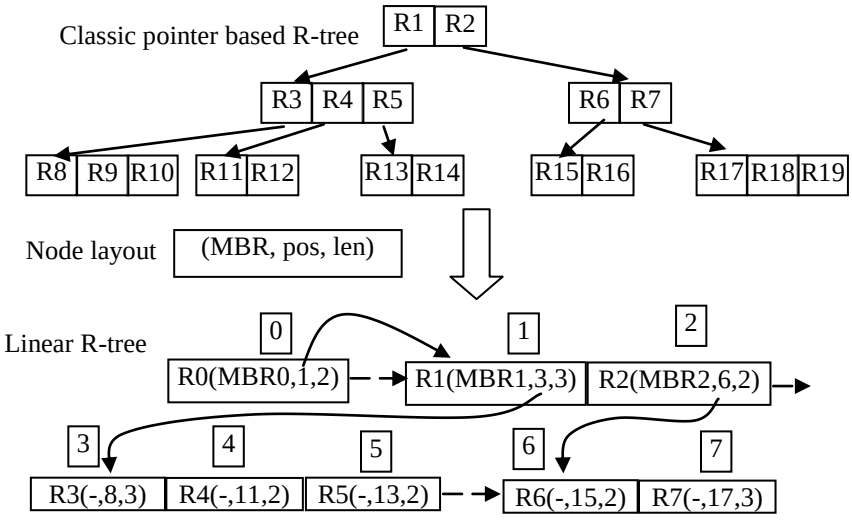


Figure 15 Illustration of Linear R-tree Node Layout

bounding rectangle of the corresponding node, *pos* and *len* are the first child position and the number of children, respectively. The tree nodes are serialized into an array based on the Breadth-First-Search (BFS) ordering. The design is illustrated in Figure 15.

Compared with a previous work reported in [63] that stored entries for all children in non-leaf nodes, our design is more memory efficient. The decision to record only the first child node position instead of recording the positions of all child nodes in our approach is to reduce memory footprint. Since sibling nodes are stored sequentially, their positions can be easily calculated by adding the offsets back to the first child node position. In addition to memory efficiency, the feature is desirable on GPUs as it facilitates parallelization by using thread identifiers as the offsets. As discussed in Section 2.2.3, an R-tree can be constructed through either dynamic insertions or bulk loading. In our targeting applications, as the datasets (such as administrative boundaries) are usually static or infrequently updated, we focus on bulk loading which allows simple and elegant implementations using parallel primitives.

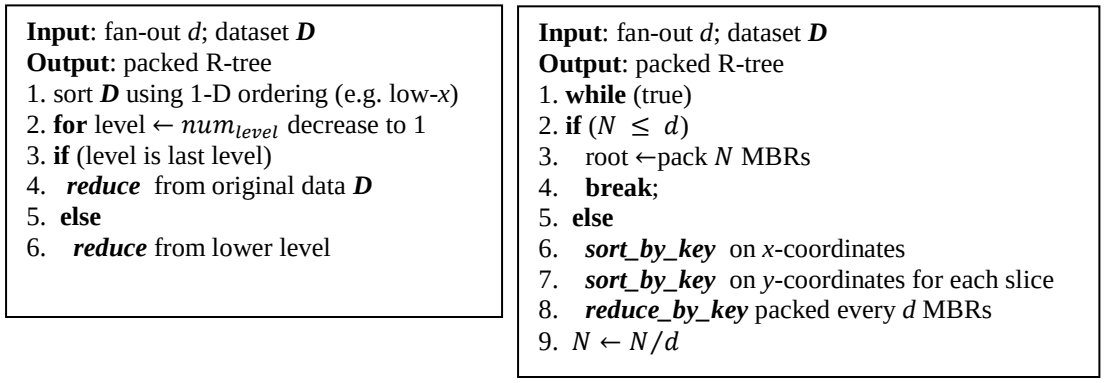


Figure 16 Parallel R-tree Bulk Loading

3.2.3.2 Parallel R-tree construction

In this study, we have developed data parallel designs based on both low-*x* packing (used in [63]) and Sort-Tile-Recursive (STR) packing [5, 55] to construct bulk-loaded R-trees. For the low-*x* packing approach, the initialization step first sorts the original data (MBRs) by applying a linear ordering schema (sort based on low-*x* in this case, other linear order may also apply). An R-tree is constructed in the main step by packing MBRs bottom-up, and the parallel design using parallel primitives is illustrated in the left part of Figure 16. Line 1 sorts the original dataset using low-*x* ordering. From Lines 2 to 6, an R-tree is iteratively packed from lower levels. In Line 4 and 6, keys with the same identifiers need to be generated every *d* items for parallel reduction purpose. The MBRs, first child positions and numbers of children are computed from the data items at the lower levels as follows. For the *d* items with a same key, the MBR for the parent node is the union of MBRs of the children nodes. For each R-tree node, the first child position (*pos*) is computed as the minimum sequential index of lower level nodes and the length (*len*) is calculated by counting the number of child nodes. Figure 17 is an example of R-tree bulk loading with fan-out set to 3. Objects (O_1, O_2, O_3, …) are first sorted by low-*x* coordinates. Then the R-tree is constructed by recursively packing from lower levels until reaching the root. For example, O_1, O_2 and O_3 are first packed and represented as {MBR_1, 0, 3} in a higher level, where MBR_1 is the union extent of O_1, O_2 and O_3, 0 is the index of O_1 and 3 represents that there are three items in this node. Similarly, {MBR_2, 3, 3} and {MBR_3, 6, 3} are generated. Finally, the root of the tree ({MBR_0, 1, 3}) is generated by packing all three nodes from the previous step.

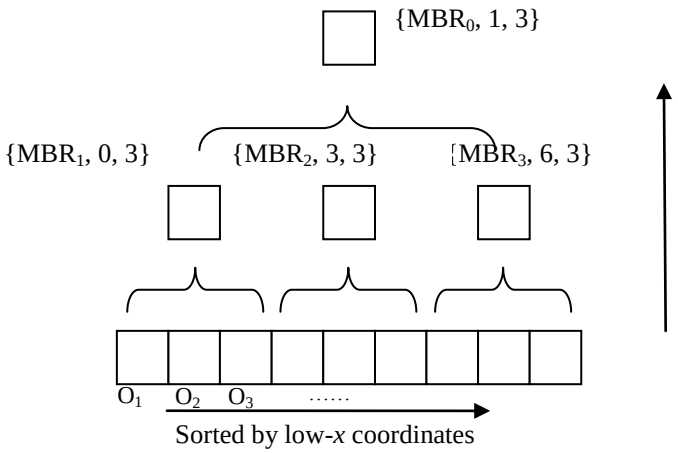


Figure 17 Low-*x* R-tree Bulk Loading Example

We note that the linear ordering of MBRs will directly impact the qualities of constructed R-trees and subsequently impact the query performance on R-trees [46, 55]. This is because spatial adjacency in 2-D may not be well preserved in 1-D, an issue that has been intensively studied in spatial databases [67]. In addition to low-*x* packing, we have also developed the STR R-tree bulk loading algorithm, which can preserve spatial locality better. The algorithm is developed using parallel primitives as follows. First, MBRs are sorted along one direction, i.e., using *x* coordinates from lower left corners, which can be implemented by using a *sort* primitive. Then the space is divided into slices according to the predefined fan-out *d*, and each slice is sorted along the other direction, such as *y*-coordinates. Finally every *d* MBRs in a slice are packed as parent nodes which will be used as the input for the next iteration. This process is iteratively executed until the root of the tree is constructed. The right part of Figure 16 outlines the STR R-tree construction algorithm. Lines 2 to 4 check whether the number of MBRs is smaller than the fan-out *d*. If this is the case, the MBRs will be packed as the root node and the

iteration is terminated. Otherwise, the MBRs are first sorted using low-*x* coordinates (Line 6), and N MBRs are divided into $\sqrt{N/d}$ slices where each slice is sorted according to low *y*-coordinates (Line 7). After sorting on each slice, parent nodes are generated via packing every d MBRs (Line 8). Finally, N/d nodes are used as the input for the next iteration (Line 9). The first sort can be easily implemented by sorting data using *x*-coordinates as the key. To implement the second sort where each slice is sorted individually, an auxiliary array is used to identify items that belong to the same slice. This is achieved by assigning the same unique identifier for all items belong to the same slice, i.e., a sequence identifier is assigned for each slice and stored in the auxiliary array. With the help of the auxiliary array, Line 7 can be accomplished by performing sort on two keys, where the primary key is *y*-coordinates and the secondary key is the unique identifiers in the auxiliary array. Line 8 is the same as the packing phase of low-*x* packing introduced previously (Lines 4 and 6 in the left of Figure 16). The difference between the two packing algorithms is that the low-*x* packing algorithm only sorts once while the STR packing algorithm requires multiple sorts at each level. Figure 18 shows a running example of the sorting and tiling process. First, all MBRs are sorted by *x* and divided into three strips (left of Figure 18). Then, within each strip, MBRs are sorted on the *y* direction (middle of Figure 18). Finally, tiles are generated by further dividing each strip as shown in the right of Figure 18. During tree construction, the same process is recursively called until the root is reached.

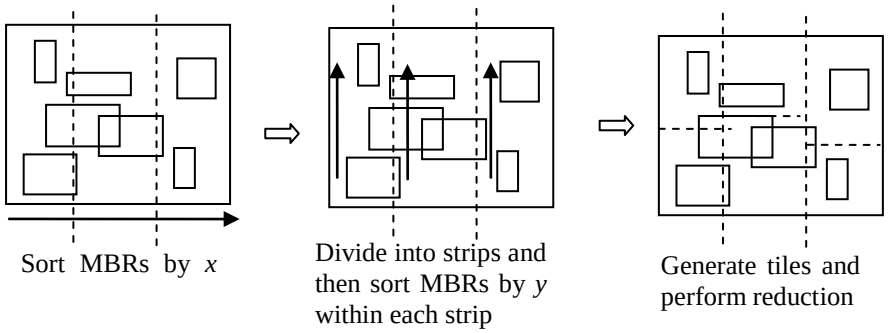


Figure 18 STR R-tree Bulk Loading Example

3.2.3.3 Parallel Batch Query on R-tree

After introducing the parallel design of R-tree construction, we next introduce our parallel design for batch spatial range queries on R-trees. As we have introduced in Section 2.2.3, Luo et al. [63] proposed a BFS based batch query technique on GPUs, where multiple queries are assigned to a block and a queue is maintained for the whole block. With such a design, a better intra-block load balance can be achieved and GPU shared memory can be used to further speed up the query processing. The authors addressed the queue overflow issue by adding another step to re-run overflowed queries repetitively until completion. However, their design was tied to specific hardware (i.e., GPU) and may not be suitable for other parallel platforms. Meanwhile, workload balance in [63] was limited to a block. In contrast, our design uses a global queue for all queries instead of multiple queues in [63], which generally leads to better load balancing. In addition, our design not only works on GPUs but also can be easily ported to other parallel platforms such as multi-core CPUs.

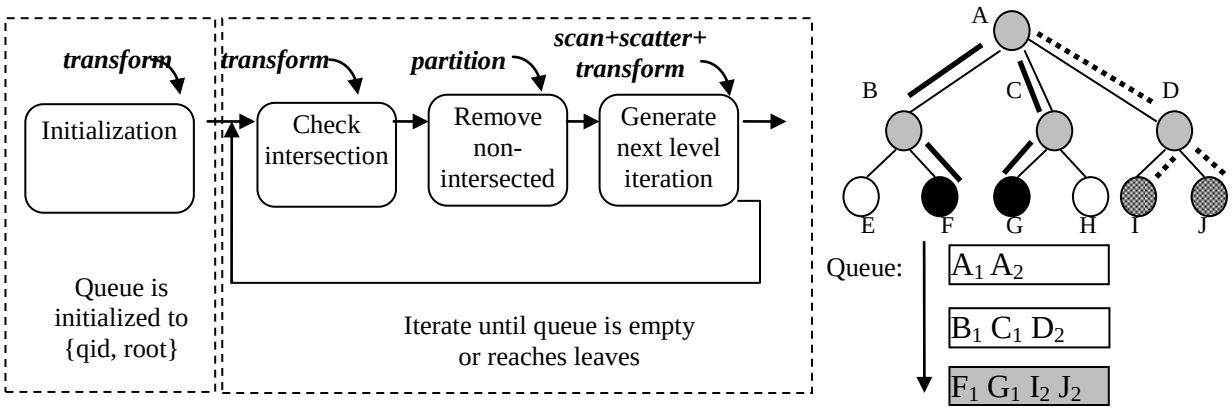


Figure 19 Parallel Primitive based BFS Batch Query

The left side of Figure 19 outlines our parallel primitives based design. First, a global queue is maintained and it is initialized using the root of the R-tree for each query. Second, all queries are checked for intersection with its corresponding R-tree node in parallel using a *transform* primitive which applies the intersection test operator for all the pairs. Third, non-intersected pairs are removed from the queue and the queue is compacted. Fourth, intersected nodes are then expanded to prepare for the next iteration. This step is a combination of several parallel primitives such as *scan, scatter* and *transform*. The iteration terminates when the queue is empty or the last level of the R-tree is reached. Finally, query results are copied from the queue to an output array. A running example is illustrated in the right side of Figure 19. Two queries and their execution traces are illustrated in bold and dashed lines, respectively. At the beginning, the queue is initialized with pairs of the root node (A) and query id (1 and 2). After that, the R-tree nodes are checked and expanded to the next level R-tree nodes (B, C and D). Finally, the iteration terminates and the queue represents query results (F_1, G_1, I_2 and J_2).

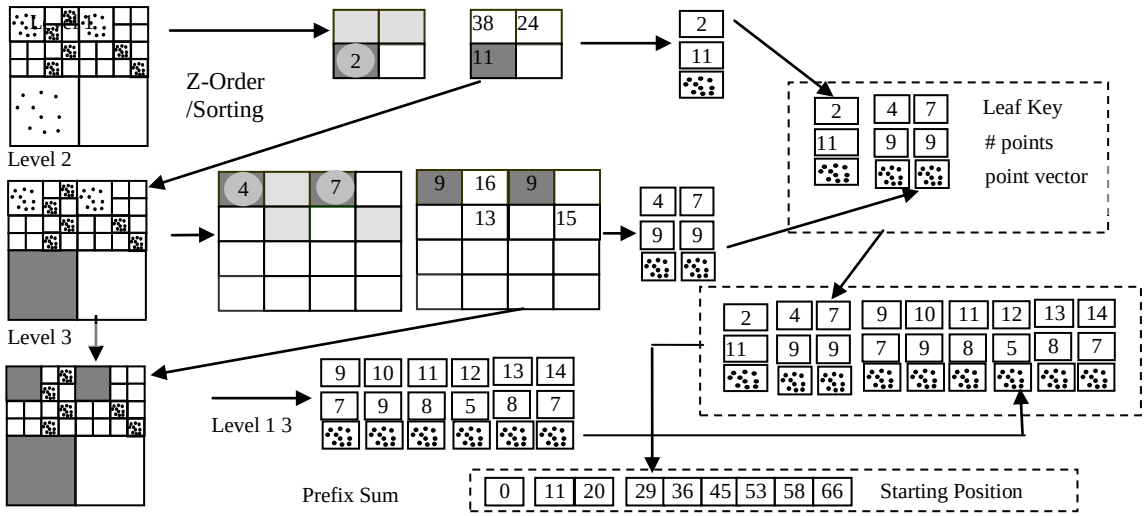


Figure 20 A Running Example to Illustrate the Process of Generating Point Quadrants

We note that there are two potential issues in our design. First, the queue is maintained globally without considering specific hardware features such as fast shared memory on GPUs. At each level, the expansion of the current nodes requires global memory accesses, which can be more expensive than accessing shared memory on GPUs and may lower its performance. Second, the parallel primitives based implementation imposes additional overhead by parallel primitive libraries when compared with using native parallel programming languages such as CUDA. However, as shown in Section 5.2.1, despite the disadvantages, the simple and portable design has achieved reasonable performance and represented a good balance between code efficiency and portability and development productivity.

3.2.4 Parallel Quadtree based Indexing for Points

Although point datasets can be indexed using parallel indexing techniques for MBRs introduced previously by treating a point as a MBR, it is not efficient for large point datasets which is typical in practice. As such, we develop a parallel Quadtree indexing technique to index large-

scale point data, which can be used to support spatial range queries and spatial joins. There are two steps in the introduced Quadtree based indexing technique for point data: step 1 generates non-leaf quadrants with each quadrant has at most *K* points, and step 2 assembles the leaf quadrants into a tree structure. Both steps are based on parallel primitives.

Input: point dataset ***P,*** max level *M*, min number of points *NP*
Output: re-arranged point dataset ***P',*** quadrant key vector ***Q,***
vector of numbers of points falling within quadrants ***Len,***
vector of numbers of starting positions of points in quadrants ***Pos***

1. for *k* from 1 to *M* levels:
2. ***Key*** ← Z-order(***P,*** *k*)
3. sort by ***Key*** on ***P***
4. reduce by ***Key*** and count number of points ***Num_Pts*** for each key
5. for each *key* in ***Key***:
6. if *num_pts* ≤ *NP*:
7. copy quadrant and points to ***P'*** and ***Q,*** and generate ***Len*** and ***Pos***
8. remove the copied subset from ***P***
9. prepare ***P*** for next iteration

Figure 21 Algorithm of Parallel Point Quadrant Generation

We present the following data parallel design for generating leaf quadrants from point dataset and the idea is illustrated in Figure 20 using an example. The strategy is to partition the point data space in a top-down, level-wise manner and identify the quadrants with a maximum of *K* points at multiple levels. While the point quadrants are being identified level-by-level, the remaining points get more clustered, the numbers of remaining points become smaller, and the data space is reduced. The process completes when either the maximum level is reached or all points have been grouped into quadrants. The maximum number of points in a quadrant (*K*) and the maximum level are set empirically by users.

The algorithm of generating point quadrants is listed in Figure 21. Starting from level 1 to *M* of the Quadtree, quadrants are recursively generated from points. Line 2 generates Z-order code as the sort key, which can use a *transform* primitive. The current level *k* is used for generating quadrant keys for the current level. For example, at the first level only the first two bits of the Z-order code are used as the key. As a result, all points within the same quadrant will have the same key and stored consecutively due to the sort in Line 3. Line 4 counts the number of points for each key using a *reduce* primitive. Line 5-8 check the counts of quadrants, and move quadrants that meet the requirement to the output vectors. After that, the dataset is compacted and prepared for the next iteration.

A complete Quadtree can be subsequently constructed from leaf quadrants using the similar layout for R-trees as introduced in Section 3.2.3.1. However, since the number of children for Quadtree is either zero or four, we do not need the *len* array that has been used in R-

Input: leaf quadrants **Qs** where each item is (*z_val, lev, p_id*)
Output: Quadtree **T**
ParallelConstructQuadtree(**Qs**):
1. *sort* **Qs** by *z_val*
2. *sort* **Qs** by *lev*
3. (*lev*, *lev_size*) = *reduce* **Qs** by *lev* //count size of quadrants at each level
4. *lev_pos* = *exlusive_scan*(*lev_size*) //get start position for each level
5. *copy* last level quadrants from **Qs** to **T**
6. *current_lev* = MAX_LEV
7. while (*current_lev* > 0):
8. *current_lev* = *current_lev* – 1
9. *transform* and *reduce* quadrants in **T** at *current_lev*+1 to *current_lev* and save in **TempQs**
10. *copy* quadrants at *current_lev* from **Qs** to **TempQs**
11. *sort* and *unique* **TempQs**
12. *copy* **TempQs** to **T**
13. return **T**

Figure 22 Parallel Quadtree Construction from Leaf Quadrants

trees. We adopt a parallel primitive based design of constructing a complete Quadtree from its leaf quadrants, as listed in Figure 22. The input (**Qs**) is a vector of leaf quadrants with their corresponding identifiers to the points and the output will be the constructed Quadtree (**T**). We use *z_val*, *lev* and *p_id* to represent Morton code, level and the corresponding point identifier respectively. At the beginning, Lines 1-2 sort leaf quadrants with their Morton codes and levels. After this step, the level boundaries are extracted in Line 3 and 4, which will be used in the following for generating non-leaf quadrant at each level. We first copy last level quadrants to the tree (Line 5), and complete the tree in a bottom up manner (Line 7-12). To generate a new level, say *current_lev*, there are two major components. One component directly comes from leaf quadrants. With the pre-generated level information at Line 3 and 4, we can easily locate leaf quadrants at *current_lev* and copy them to a temporary space (**TempQs**). The other component should come from the reduction of lower level quadrants, in other words, the quadrants at *current_lev*+1. Those quadrants then are appended to **TempQs**. We note that, to maintain the link between two consecutive levels, the first child position (*fc*) must be set appropriately. This is achieved by performing a *reduce* operation where four child positions that belong to the same parent are applied by a *min* operator. The last step is to copy the **TempQs** to the tree structure **T** (Line 12). The iteration continues on a higher level until the root of the tree is reached.

Using the constructed Quadtree, the batch query processing is almost identical to using R-tree (Section 3.2.3.3) except that the MBR of a quadrant is implicitly stored in the format of a Morton code. We thus skip the details of spatial query processing on Quadtrees for point data.

3.3 Multi-Node Distributed Spatial Indexing

To efficiently process large-scale spatial datasets on multiple machines (multi-node) with reasonable load balancing, one of the most important techniques is spatial partition to divide large-scale datasets into small pieces and each piece can be processed on a single machine. We have developed distributed spatial indexing based on spatial partitioning in order to support efficient large-scale spatial data processing on multi-node environments. The distributed spatial indexing structure is illustrated in Figure 23. The structure consists of an index file and a partitioned dataset, which is similar to VegaGiStore [108] and the distributed index in SpatialHadoop [24].

Unlike existing works that are tightly coupled with their execution environment, we design the distributed indexing as a separate module which is independent from execution environments such as Hadoop. As shown in the figure, the index is stored as a separate file without adding additional information in the original dataset. The dataset is only re-organized according to one of the spatial partition strategies that will be introduced later. In the index file, we store metadata about the indexed dataset. For each partition we store the MBR of the partition as well as other metadata. A link is maintained for each partition in order to access the corresponding data block, which can be either a file location or an offset in the data file. Our distributed index works for any partition strategies, which is different from VegaGiStore that can only use Quadtree-like partition [108]. Meanwhile, we design the distributed indexing to be platform independent and the index is stored as a separate file so that other systems without the indexing module can still work on the raw dataset. The design is different from SpatialHadoop where local index is saved into the partitions, which makes it incompatible with other systems.

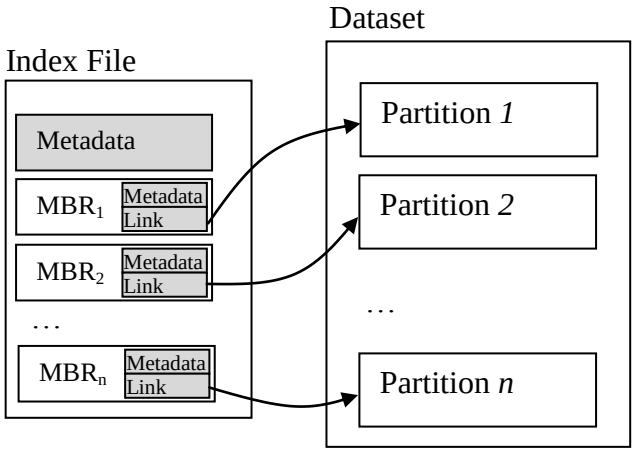


Figure 23 Distributed Spatial Indexing Structure

In this work, we have designed and implemented all the three partition strategies introduced in Section 2.2.4 using data-parallel primitives. As data-parallel primitives are well supported by parallel libraries as well as Big Data systems such as Spark, our implementations are easy to be implemented. The data-parallel designs are also applied to shared-memory systems such as multi-core CPUs and GPUs with data-parallel primitive libraries. Although Hadoop-GIS has a similar effort on spatial partitioning [96], their implementations were sequential and the design has not been developed for data-parallel environments. Unlike SpatialHadoop that stores the local index in each partition, we consider either storing local indexes to the index file or completely removing local indexes for partitions. The design of storing local indexes in the separated index file provides compatibility on different systems. On the contrary, datasets indexed by SpatialHadoop cannot be processed by other systems if the dedicated data loader is not implemented. Besides, local indexes may not be useful if random data access is not supported (such as functional operators in Apache Spark). In this case, using local indexes imposes additional IO overhead without benefiting system performance. On the other hand,

hierarchical spatial indexing structures such as R-tree may not be efficient on current generation of hardware due to irregular memory access and synchronization. Since a partition can be completely loaded in memory to process, performing parallel scan on the whole partition instead of traversing index can potentially be more efficient, especially when caching is taken into consideration. An alternative solution is to build on-demand local index using bulk-loading for expensive spatial operators such as spatial join. Our design guarantees the separation of indexing structure and original dataset, and only necessary data re-ordering is applied to the original dataset.

3.4 Summary

In this chapter, we have introduced parallel designs of spatial indexing techniques, including both space- and data-oriented indexing structures. Data-parallel designs of space-oriented data structures such as Grid-file and Quadtree have been presented. Since these two indexing structures regularly decompose the space to be indexed, the spatial problem (spatial query) can be transformed into a non-spatial problem (binary search) that is suitable for parallelization. For objects overlapping with indexing unit boundaries, such as grid cell boundaries in grid-file indexing, they are duplicated in each overlapping grid cell in order to ensure complete query results. Therefore, additional duplication removal step is used to generate unique results. Memory footprint can be a bottleneck because redundant information is required to store. The data-oriented partition can solve the duplication issue more effectively because the boundaries can be computed from the data. However, it brings challenges for parallelization due to irregular partitioning. We have developed data-parallel R-tree techniques, for both index construction and batch query processing. The data-parallel designs are GPU friendly so that we can take

advantages of promising hardware accelerator. In additional to parallel indexing on a single node, we have also discussed the design of distributed indexing that can be applied to distributed environments especially for big data platforms. Different from existing works such as SpatialHadoop, our design emphasizes on not only efficiency but also compatibility. Meanwhile, we have discussed parallel batch query processing using our parallelized spatial indexing structures in this chapter, which can be applied in spatial join processing in the next chapter.

Chapter 4 Parallel and Distributed Spatial Join

To develop efficient spatial join designs on modern parallel and distributed platforms, we break down the problem into two levels. First, we develop parallel techniques that are used for accelerating single-node spatial join, which are able to exploit parallel computing power on a single machine. At the second level, we design spatial join techniques for distributed environments to achieve scalability. By combining the two levels of parallelism, we are able to perform spatial join effectively at very large scales.

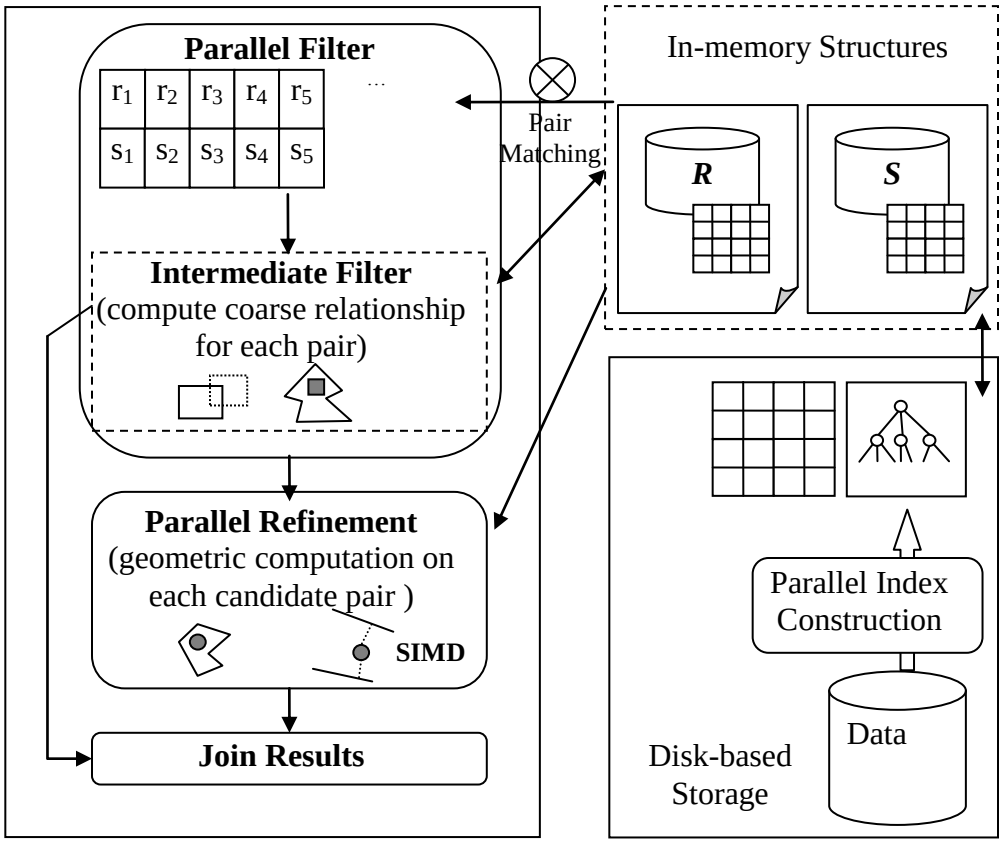


Figure 24 Single Node Parallel Spatial

4.1 Single-Node Parallel Spatial Join

As introduced in Section 2.3, a spatial join typically has two phases, i.e., *filter* and *refinement*. When a spatial join is performed on a single node, the filter phase first generates candidate pairs by using approximated representations and the refinement phase completely removes false positives to produce exact results. The spatial filter phase shares several similarities with batch-query on spatial indexes as discussed previously. However, it is possible that none of the input datasets in a spatial join is indexed. In this case, a spatial join needs to choose a proper filter strategy, including building indexes on-the-fly, to join the data items in the input datasets efficiently. In addition, while the number of spatial queries (represented as MBRs) in a batch can be large, it is typically smaller than the number of data items of the input datasets in a spatial join. More importantly, spatial refinement in a spatial join can dominate the whole process and its performance is critical for the end-to-end performance. As such, additional techniques sitting between filter and refinement phases that can further improve pruning power and reduce the number of tests of spatial predicates in the refinement phase are preferable. Although the spatial indexing and query processing techniques that we have developed in Section 3.2 are data-parallel and efficient, we would like to investigate on more techniques that can potentially improve spatial joins on large datasets and improve single-node efficiency for spatial join. The framework of our parallel spatial join technique on a single node is illustrated in Figure 24.

4.1.1 Parallel Spatial Filtering

We have developed lightweight on-the-fly spatial indexing for spatial join that involves point datasets, such as point-in-polygon test based spatial join. Recent studies [73, 87] have shown that using non-hieratical and simple spatial indexes on modern parallel hardware may produce better

performance than using classic hieratical spatial indexes (e.g., R-tree). Given that spatial join between a large and dynamic point dataset (e.g., taxi trip locations) and a relatively small and static polygon/polyline dataset (e.g. administrative zones) based on point-in-polygon test is one of the most popular types of spatial join, we next introduce a lightweight on-the-fly indexing technique for a large point dataset to be joined with a polygon/polyline dataset that is pre-indexed using a grid-file (described in Section 3.2.2).

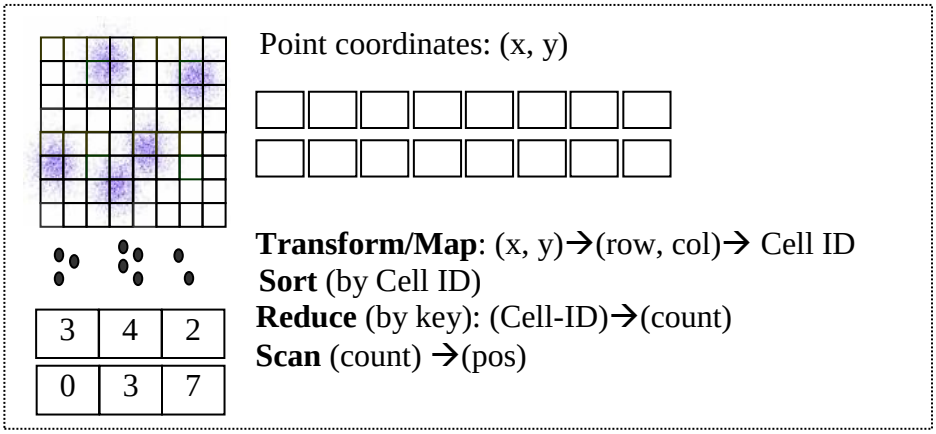


Figure 25 Light-weight Indexing for Point Dataset

Assuming a grid-file has been created by indexing the input polygon/polyline dataset, the idea is to create a grid-file for the input point dataset, which may have a high update frequency and may not be previously indexed. Clearly, it is desirable to use the same grid-file configuration of the input polygon/polyline dataset for the input point dataset, which is possible in spatial join as we are building a grid-file index on demand. The design of the lightweight indexing technique for point data using parallel primitives is illustrated in Figure 25. The *transform* primitive generates grid cell identifiers for all points; the *sort* primitive sorts points based on the cell IDs;

the *reduce (by key)* primitive counts the number of points within each grid cell; and finally the *(exclusive) scan* primitive computes the prefix-sums of the numbers of points in all grid cells which are the starting positions of the points in the sorted point data vector.

Compared with Quadtree based point indexing technique presented in Section 3.2.4, the design is indeed lightweight which makes it desirable for spatial joins. However, this is at the expense that the number of points in a cell can be potentially unbounded and may incur load unbalance in spatial refinement when the points in a cell is assigned to a processing unit in a naïve way. Fortunately, parallel libraries such as TBB on multi-core CPUs can tolerate load unbalancing to a certain degree by using algorithms such as work stealing [64]. Similarly, CUDA computing model also tolerates load unbalancing to a certain degree at the computing block level as GPU hardware assigns computing blocks to GPU cores in the units of warps dynamically. We plan to investigate techniques that can mitigate load unbalancing, such as merging cells with too few points and splitting cells with too many points.

To further improve the efficiency of the point-in-polygon test based spatial join, we have added an intermediate step between the spatial filter phase (based on grid cell matching) and spatial refinement (based on point-in-polygon test) using cell-in-polygon test. The idea is illustrated in Figure 26. The motivation is that, if a cell is completely within/outside a polygon, then all the points that are indexed by the cell will be completely within/outside the polygon without needing performing the expensive point-in-polygon tests for the points individually. If the number of the points in the cell is large, it is likely that the overall performance can be significantly improved. For example, in the right side of Figure 26, point-in-polygon tests in cells

A, A’ and B can be saved since they are either completely outside or inside the polygon. We note that cell-in-polygon test can also adopt a data parallel design in a way similar to the design of parallelizing the point-in-polygon test design to be described next. We note that similar ideas can be also applied to other types of spatial joins which are left for our future work.

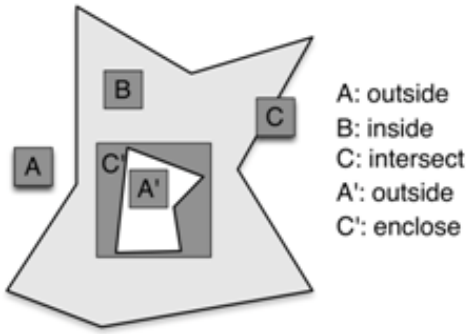


Figure 26 Cell-to-polygon Relationship

4.1.2 Parallel Refinement

The results of the filter phase are candidate pairs whose MBRs meet the spatial relationship but with false positives. Thus, a refinement phase is used to completely remove the false positives and generate the final results. The refinement phase usually involves geometric computations, such as point-in-polygon test, to determine the exact spatial relationship of candidate pairs. The geometry operations that we will be focusing on include distance based and topology based operations. The distance based operations are mainly used for nearest neighbor search based spatial joins that involve distance calculation. For topology based operations, we currently focus on intersection test based spatial join, such as point-in-polygon test.

Geometry operations have been well studied in computational geometry and implemented in several general-purpose geometry libraries such as JTS[19] and GEOS[20]. However, to our best knowledge, there is no existing geometry library that can fully take advantages of SIMD units on CPUs as well as GPUs. Unfortunately, using a general-purpose geometry library such as GEOS to perform geometry operations is very slow based on our in-house benchmarks. Thus, we have developed a specialized geometry engine that is parallelized on both CPUs and GPUs based on our columnar spatial data layout introduced in Section 3.2.1. The engine supports major spatial data types (including point, polyline and polygon and related distance based and topology based operations). The major challenge of developing the geometry engine is to design data-parallel geometry algorithms that can exploit SIMD parallel computing power. In the refinement phase of spatial join, the computation usually performs on a set of candidate pairs instead of a single pair. As such, we design the geometry engine to process a geometry operation in batches that can be mapped to multi-core CPUs (with VPUs) and GPUs for efficient SIMD processing. We next introduce our design using point-in-polygon test operation as an example. Other operations such as distance calculation of two spatial objects for nearest neighbor search can follow a similar design.

[19] http://www.vividsolutions.com/jts/JTSHome.htm

[20] http://trac.osgeo.org/geos/

During the refinement phase of point-in-polygon test based spatial join, we assign each pair of point-in-polygon test to one SIMD execution unit (i.e., thread in GPU and VPU lane in CPU). Using the classic ray-casting algorithm for point-in-polygon test [40], a point loops through all the vertices of its paired polygon on each SIMD execution unit. As nearby points have similar spatial coordinates, it is very likely that all execution units on all VPU lanes in a CPU core or a computing block on a GPU follow a same path. As discussed in the next two paragraphs, the design is efficient on CPUs due to cache locality and efficient on GPUs due to coalesced memory accesses. Although there exist point-in-polygon test algorithms in the complexity of O(*log n*) or even O(1) [40, 41, 56, 110], we argue that the ray-casting algorithm does not require additional pre- and post-processing on polygons, and the simplicity of its

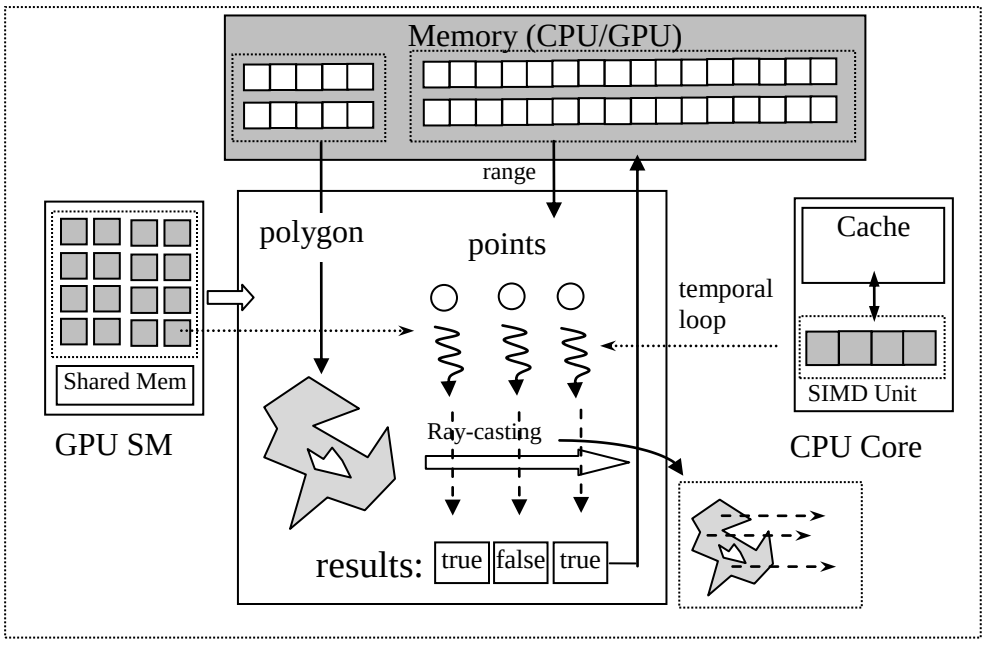


Figure 27 Point-in-polygon Refinement on CPU and GPU

implementation makes it robust. Meanwhile, the implementation of our point-in-polygon test directly manipulates data items in SoA, which is very efficient comparing with existing libraries that usually have significant abstraction overheads and are not cache friendly due to excessive dynamic memory allocations.

The parallel designs of point-in-polygon test operation on both multi-core CPUs with VPUs and GPUs are further illustrated in Figure 27. For GPUs, we assign a group of points to a GPU computing block, in which all points within the group perform point-in-polygon tests on the same polygon. Each GPU thread loads a point and loop through all vertices of the targeting polygon in a lockstep manner. If the test result is positive, its corresponding indicator is set and saved to GPU global memory. Since points are stored consecutively, the global memory access is perfectly coalesced. As for polygons vertices, since all threads in a computing block access the same polygon vertex at a time, the vertex data can be broadcast to all threads in the warps of the computing block by GPU memory controller hardware, which is highly efficient on GPUs. The multi-core CPU design is very similar to the GPU design, where each test is assigned to a SIMD lane in VPUs instead of a thread in GPU. Since all SIMD lanes within the VPU of a CPU core are accessing vertices of the same polygon in the same order, it is efficient on memory accesses.

The difference between GPU and multi-core CPU for the refinement is mainly on task decomposition and execution. A point-in-polygon test task on multi-core CPUs is divided into subtasks based on ranges of points and a micro batch with size equals to the number of SIMD lanes is assigned to the VPU on the CPU core to loop through all the points in the range. On GPUs, a range of points is assigned to a thread block for parallel processing and the GPU

hardware maps the threads to GPU cores in warps (Section 2.1.1.2). While CPUs may cache both points and polygon vertices to reduce memory access costs, GPUs mainly rely on coalesced memory accesses (for points) and broadcast memory accesses (for polygon vertices that are shared) among threads in warps to hide memory access latency.

4.2 Multi-Node Distributed Spatial Join

To perform spatial join on very large datasets, especially when the sizes of data exceed the capacity of a single machine, we need to develop efficient distributed spatial join techniques for multi-node computing environments, i.e., a cluster with multiple machines. We have developed two distributed spatial join designs based on the characteristics of input datasets. When both datasets are very large and at a similar scale, we call the two datasets *symmetric*. To process spatial joins on symmetric datasets (or *symmetric spatial join*), we have developed spatial partition based spatial join techniques, where data are divided based on a predefined spatial partition schema and processed individually in distributed computing nodes. However, the process of generating spatial partitions can be very expensive if the datasets are large. On the other hand, we have observed that in many spatial join applications the input datasets are *asymmetric*. This means, one of the two input datasets is relatively small comparing with the other one. For example, a point-in-polygon test based spatial join application involves a large number of GPS locations and a moderate size of administrative zone boundaries. As one side of the join inputs (boundaries) is relatively small comparing with the other side (GPS locations), we term the spatial join as *asymmetric spatial join*. For this type of spatial join, instead of performing expensive spatial partition that is necessary in spatial partition based spatial join, we have developed a more efficient approach by broadcasting the small dataset to all the partitions

of the large dataset for distributed executions. In this section, we will introduce spatial join implementations that take advantage of the state-of-the-art Big Data technologies, including several prototype systems.

4.2.1 Spatial Partition based Spatial Join

We have developed a spatial partition based spatial join technique to process symmetric spatial joins on multi-node platforms. The parallelization on a multi-node environment is different from single-node parallelization. For example, while random access is well supported on a single machine because of shared-memory architectures within single computing nodes, it is very expensive on shared-nothing cluster computers that involve data communication among distributed nodes. When designing parallel spatial join techniques on multiple computing nodes, it is necessary to minimize the expensive communication cost in order to improve end-to-end performance. On the other hand, in parallel computing, the overall performance is usually dominated by stragglers (slow nodes). A good parallelization design has to minimize the effects from stragglers. Therefore, the basic idea of our spatial partition based spatial join technique is: divide the spatial join task into small (nearly) equal-sized and independent subtasks and process those small tasks in parallel efficiently. The technical challenges are as follows: 1) how to divide a spatial join task into small non-overlapping tasks that can run in parallel with low communication cost, 2) how to divide a spatial join task in a way that achieves better load balance. We introduce spatial partition based spatial join techniques to address those challenges.

Spatial partition based spatial join is designed in two phases, i.e., the partition phase and the local spatial join phase. In the partition phase, a partition schema is computed based on the

spatial distribution of either the whole or a sample of the input dataset and the input data are subsequently partitioned according to the partition schema. If both datasets have already been partitioned using one of the previous introduced strategies (such as FGP, STP or BSP in Section 2.2.4), there are two methods to accomplish the partition phase. The first method is to repartition one dataset according to the existing partition boundaries from the other dataset. Another method is to match partition boundaries from both datasets and each matched pair is considered as a virtual partition that will be assigned to a computing node. After the partition phase, each partition contains objects from both sides and local parallel spatial join is performed using the techniques developed for single-nodes as introduced previously. By this means, we are able to achieve two levels of parallelism, i.e., inter-node parallelism and intra-node parallelism. As intra-node parallelization has been discussed in the previous sections, we focus on inter-node parallelization in this section.

An example of spatial partition based spatial join on non-indexed datasets is illustrated in the left part of Figure 28. First, a partition schema is generated by sampling the input datasets (A and B). After that, A and B are partitioned by the schema and each partition holds subsets of the original datasets, e.g., A_1 and B_1 are in partition 1. Finally, partitions (Partition 1, 2 and 3 in the figure) are assigned to a computing node for local spatial join processing. If input datasets have already been partitioned (indexed), an alternative approach is to match existing partitions instead of performing repartition. The approach is illustrated in the right part of Figure 28 which is almost identical to the previous one except that the partition phase is different. In this approach, instead of performing repartition in the first method, partition boundaries from both datasets are matched. Then, matched pairs are assigned to computing nodes. As discussed in previous works

[81, 96], a good partition schema may result in better performance. In this work, we have developed data-parallel designs and implementations of the three spatial partition techniques introduced in Section 2.2.4 for our partition based spatial join on modern parallel and distributed platforms. To our knowledge, this has not been addressed in previous works.

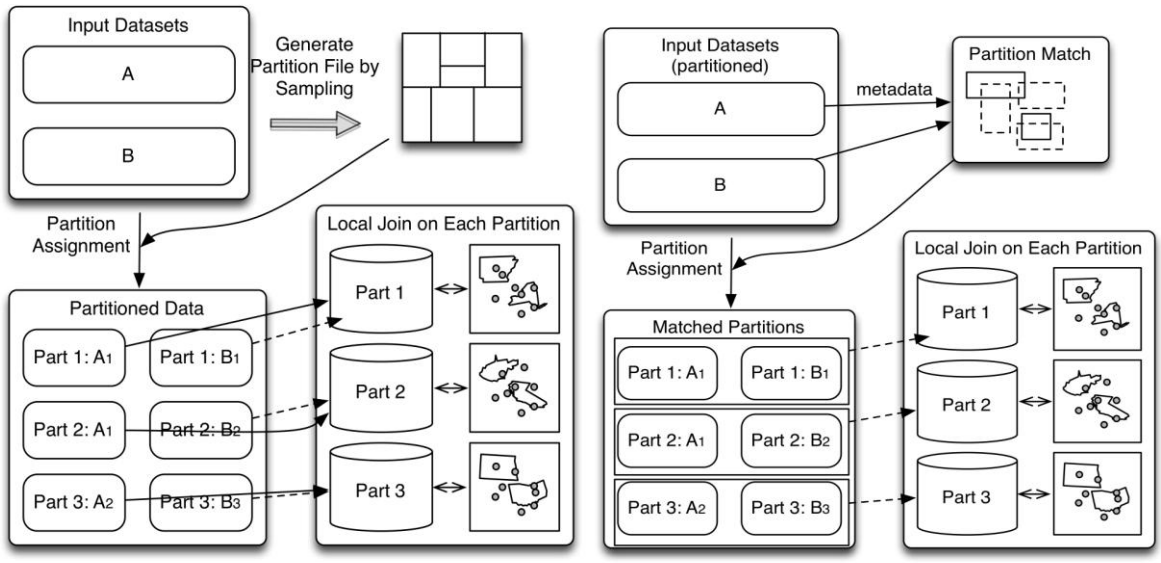


Figure 28 Spatial Partition based Spatial Join

In the first method, after an on-demand partition schema is generated, both input datasets need to be shuffled based on the schema so that local spatial join within each partition can be performed. Towards this end, each data object will be assigned a partition id based on the partition schema. For spatial objects with zero extent (such as points), the one-to-one correspondence is easy to calculate. However, for those spatial objects with non-zero extent (e.g., polygons and polylines), when they are on the partition boundaries, one object can intersect with multiple partitions and the object needs to be duplicated for each partition it intersects. When a spatial join involves buffered search, such as nearest neighbor search within a defined buffer

radius, a partition should include not only objects intersect with it but also objects that intersect with the buffered region (derived by expanding the object with the buffer radius). For FGP, the partition id can be directly calculated from the predefined grid size. However, this is not straightforward for other partition techniques because their partition boundaries can be irregular. We create a spatial index on the partition boundaries (e.g., using R-tree) and perform query processing for each data item so that the corresponding partition ids can be assigned. Since spatial objects are possibly duplicated in the process, an additional post-processing is required to remove the duplication and the easiest way is to sort. As both sort and scan can be performed on modern parallel hardware efficiently (in the orders of hundreds of millions per second), we sort the combined results and remove duplication via a full parallel scan on the results to reduce implementation complexity.

In the second method where partitions are pre-generated, we assume partition boundaries

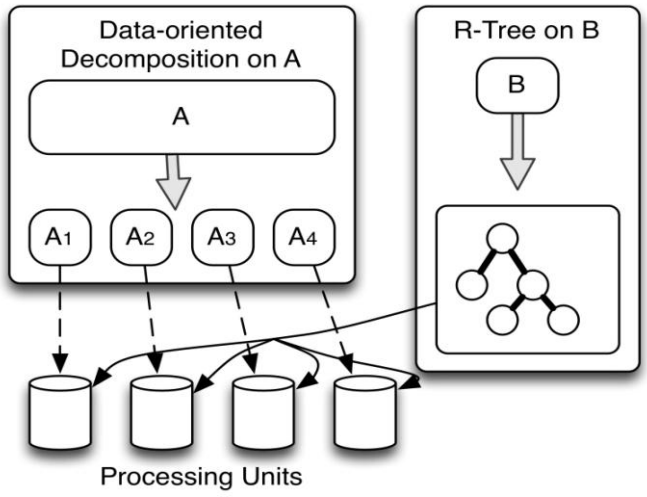


Figure 29 Broadcast based Spatial Join

are saved as metadata. In the partition matching step, metadata (partition boundaries) from both partitioned datasets are loaded and matched. Usually, a master node is responsible for performing the partition matching and a list of matched pairs will be maintained. Since the number of partitions is relatively small, we can apply single-node parallel spatial join techniques to generate the matched pairs. Once the list has been generated, computing nodes can process on the partition pairs and generate the final results. For each partition pair, we apply the single-node parallel spatial join technique as we have adopted in the previous approach.

4.2.2 Broadcast based Spatial Join

In spatial partition based spatial join, the partition phase can be very expensive due to data re-ordering as well as the additional partition matching phase. Transferring large amount of data may also degrade the overall performance significantly. This motivates us to develop an efficient spatial join technique for asymmetric spatial joins. Considering a spatial join whose left side is a large point dataset and the right side is a moderately sized polyline or polygon dataset, we can broadcast the right side to all the partitions of the left side and perform spatial join locally for better efficiency. The assumption for the broadcast based spatial join is that the small dataset can be fit in the memory of each machine which is typically valid in many applications. For example, a dataset of administrative boundaries of a city is usually in the order of tens of thousands and the data volume is no more than tens of megabytes, which can be easily stored in the main memory of current commodity computers.

Our broadcast based spatial join technique 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. In the second step, the large dataset is loaded from a distributed file system (such as HDFS) and equally distributed among the distributed nodes using range partition. Each node then performs local spatial join on its own portion of the large dataset. As the geometry objects of the small dataset are stored locally, the refinement phase can be performed without additional data transfer. Figure 29 provides an example of broadcast based spatial join. In the example, the small dataset B as well as an on-demand R-tree index is broadcast to all computing nodes. On the other side, the large dataset A is divided into chunks and processed independently by all computing nodes.

The small dataset as well as the on-demand spatial index are read-only during the whole process. Therefore, no synchronization is involved and each local spatial join can run independently. Since each data item in the large dataset performs query on the same small dataset, the runtime of query data item is roughly the same during the filter phase. However, for the refinement phase, the workload can be very different because the intensity of geometry computation varies across partitions. One solution to address this problem is to adjust workload for each cluster node by using proper selectivity estimation metrics. By avoiding the expensive data re-ordering and spatial partition, broadcast based spatial joins for asymmetric datasets can potentially achieve much better performance than spatial partition based spatial joins. Furthermore, since no additional phase to remove duplication is needed, the already reduced workload is likely to be balanced, which is desirable. To sum up, the key advantage of broadcast based spatial join is avoiding expensive overheads of spatial partitioning and data re-ordering

while the major disadvantage is that broadcast based spatial join requires larger memory and may not be applicable for joining two large datasets.

4.3 Large-Scale Spatial Data Processing Prototype Systems

To demonstrate the feasibility and effectiveness of our parallel designs introduced previously, we have developed three prototype systems based on state-of-the-art Big Data technologies. The first two prototype systems, called SpatialSpark and ISP, are based on Spark [106] and Impala [14], respectively. The third one, called LDE, is a light-weight distributed processing engine, which does not rely on existing Big Data platforms (except HDFS that is used for distributed storage). We will introduce the details of the three prototype systems in the following.

4.3.1 SpatialSpark

Based on our designs, we have initiated an effort to develop efficient big spatial data processing engine using Spark, namely *SpatialSpark*. In SpatialSpark, we have implemented both broadcast and spatial partition based spatial joins. Since Spark is written in Scala, most of Java libraries can be used without any changes. Thus we could reuse the popular JTS library for spatial refinement. For example, testing whether two geometric objects satisfy a certain spatial relationship (e.g., point-in-polygon) or calculating a certain metric between two geometric objects (e.g., Euclidian distance). In addition to utilizing finer grained data parallelism to achieve higher performance, as all the intermediate data are memory-resident in Spark, excessive disk I/Os can be minimized which is a key to achieve the desired high-performance. For geometry representation, we choose WKT format for storing geometry data in HDFS, which is simple and can be stored as native string type.

For broadcast based spatial join, we take advantage of the broadcast mechanism in Spark, which can send a broadcast variable (which can be a dataset) efficiently to all computing nodes. JTS library is used to generate R-tree from the small input dataset and the geometries are associated with the leaf nodes of the R-tree. A broadcast variable is created from the generated R-tree, which can be accessed by all computing nodes. For large datasets, each data item performs its local spatial join individually. We use R-tree batch query to generate candidate pairs and all queries are executed in parallel. The spatial refinement phase also uses JTS library to evaluate the spatial relationship in the join for each candidate pair.

The spatial partition based spatial join is more complex than the broadcast based spatial join in SpatialSpark. We have implemented all the three partition strategies introduced previously (Section 2.2.4) with both serial and parallel version variations on Spark. For fixed-grid partition, the partition boundaries can be directly calculated based on the extent and grid partition size. The partition assignment phase can be realized by simply assigning each data item

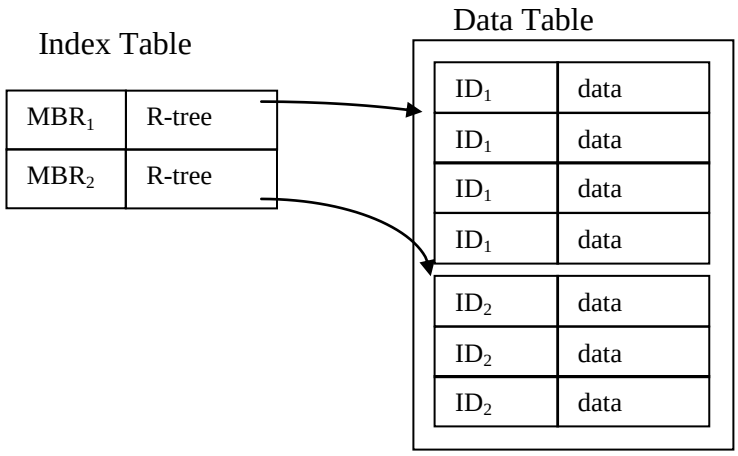


Figure 30 Table Layout in Spark SQL

with a partition id using the built-in Scala *map* primitive. For the other two partition techniques, after an R-tree is constructed, each item queries the R-tree in parallel to compute its partition ids Based on the partition ids, all data items are shuffled using the built-in *join* primitive. The partition assignment and data shuffle steps are typically time consuming due to the expensive data re-ordering as discussed previously (Section 2.2.4). After the shuffle phase, each partition contains two lists of spatial objects. Since the two lists are not indexed, we create an on-demand R-tree on one side and perform batch queries using the data items in the other side, for all partitions in parallel. This step can also be replaced with a local nested loop join or a plane-sweep join. Each local spatial join is assigned to a single thread that runs sequentially. Finally, the output is combined and saved to HDFS.

Another implementation of partition based spatial join on top of the new Spark SQL[21]

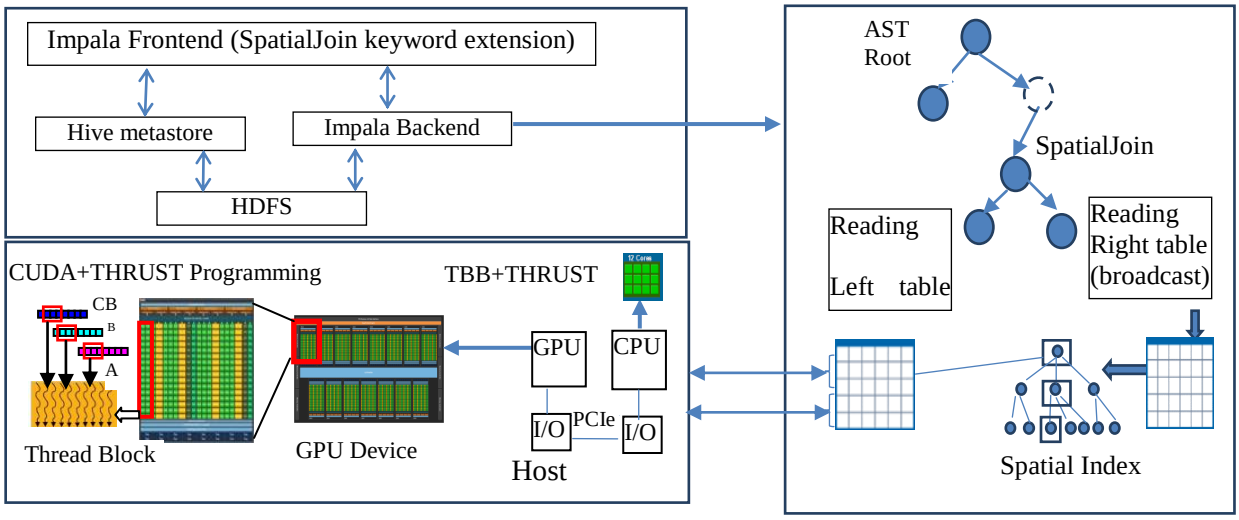


Figure 31 Spatial Join in ISP

[21] http://spark.apache.org/sql/

module using its DataFrame APIs. Instead of generating on-demand partitions and spatial indexes, spatial partitions and indexes are constructed and materialized as a separate table. Such design will be more useful if the spatial dataset will be reused for multiple queries and join processing. We adopt a two-level indexing structure where the first level partitions the dataset and the second level uses R-tree for each partition. The index structure is maintained as a table in Spark SQL. Each row of the table represents a spatial partition including a MBR for the whole partition and an R-tree for all objects belonging to this partition. The leaf nodes of the R-tree are identifiers that can be linked to the original dataset to retrieve the exact geometry representations. An example of the table layout is shown in Figure 30. In the figure, two partitions are stored in the index table where each row contains the MBR of the partition as well as an R-tree for all objects within the partition. The two tables are linked through partition identifiers. During the spatial join processing, the spatial indexes are first loaded and matched for the filter phase. After that, a list of candidate pairs is generated for the refinement phase, and each pair consists of two identifiers from both sides. The refinement phase performs a three-way join and exact geometry representations are retrieved for geometric operation. The benefit of using DataFrame is to take advantage of the optimizer and runtime code generation modules in Spark SQL, which can produce better performance than the raw RDD operations. In order to perform exact geometry refinement, intermediate results of the three-way join need to keep all geometry representations. When the sizes of joining geometry objects are getting large, the intermediate results of the three-way join can be very large due to duplication (a record from one side may have multiple join candidates from the other side), and they may exceed memory

capacity in processing computing nodes. When memory capacity is exceeded, Spark runtime will spill data from memory to disk, which can hurt performance significantly.

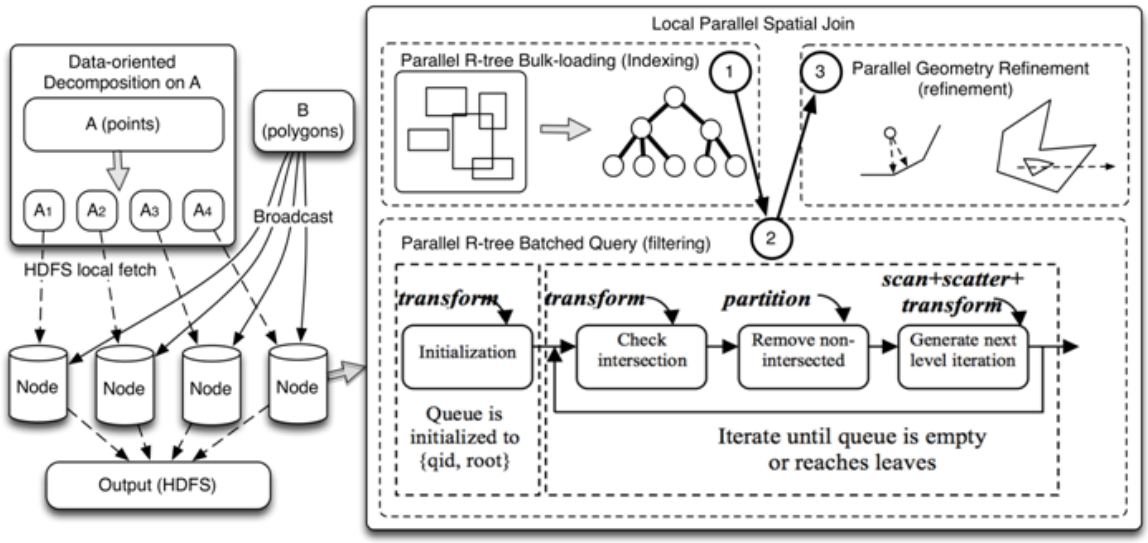


Figure 32 Point-in-polygon test based Spatial Join on ISP

As SpatialSpark is functionally equivalent to several existing packages such as Hadoop-GIS and SpatialHadoop, it is also desirable to evaluate and compare the performance among the three platforms. We have conducted performance study for several real world spatial applications to gain insights. We have also tested the scalability of SpatialSpark in the Cloud to demonstrate its capability in processing large-scale datasets. The results are provided in Section 5.3.1.

4.3.2 ISP

In additional to SpatialSpark, we have also implemented broadcast-based spatial join on Impala which is another leading in-memory processing engine. The prototype system is called ISP, including a multi-core CPU version (ISP-MC) and a GPU version (ISP-GPU). Unlike Spark, Impala query processing backend is implemented using C++. As such, it is ideal to serve as the

base for further extensions when performance is critical. In particular, as currently Java has very limited support for exploiting SIMD computing power on either CPUs or GPUs, C/C++ language interfaces might be the most viable option to effectively utilize hardware accelerations. ISP is designed to fully take advantage of hardware accelerators. Figure 31 shows the architectural design of ISP. First, we have modified the Abstract Syntax Tree (AST) module of Impala frontend to support spatial query syntax. Second, we represent the geometry of spatial datasets as strings to support spatial data accesses in Impala (as in SpatialSpark) and prepare necessary data structures for GPU based spatial query processing. Third, we integrate our single-node GPU-based spatial data management techniques with Impala to support large-scale spatial data processing on GPU-equipped clusters. We currently have implemented broadcast based spatial join due to its similarity with existing relational hash join implementation in Impala. For spatial partition based spatial join, we found its implementation using existing infrastructure is quite challenging. Unlike Spark that provides convenient parallel primitives, Impala is an end-to-end system which makes it difficult to build custom applications. Although it is technically possible to implement partition based spatial join on top of Impala, the implementation will be tied to a specific version of Impala which makes it less attractive for general use.

We next present a detailed design of the point-in-polygon test based spatial join accelerated by GPUs in ISP. In this design, we have developed broadcast based spatial join introduced previously. The process of a point-in-polygon test based spatial join using R-tree in ISP-GPU is illustrated in Figure 32. During the spatial join, the large table is first divided into row batches where each row batch consists of multiple rows and is processed on an Impala instance. Then, the small table is broadcast to all Impala instances and an on-demand R-tree is

created from the small table in an instance. We note that retrieving the small table from HDFS can be efficiently done using multi-threaded I/O supported by Impala. Meanwhile, the on-demand R-tree can adopt our parallel design introduced in Section 3.2.3. After the broadcast step, we iterate through all the row batches that are assigned to an Impala instance to perform local spatial join.

For each row batch, we use GPUs to parallelize tuple evaluations. Non-spatial sub-expressions are evaluated first on CPUs before the spatial query is evaluated on GPUs using the on-demand R-tree. This is because spatial operations are typically more expensive and can benefit from filtering based on non-spatial criteria, in addition to GPU hardware accelerations of floating point computation. The geometry of a whole row batch is transferred to GPUs for parallel query against the GPU based R-tree built in the broadcast step. The query result is then transferred back to CPUs in the form of a vector of identifier pairs. Finally, tuples of the big table and the small table are located based on the identifier pairs and they are concatenated (possibly after applying a projection operator) before written to an output tuple buffer. The buffer will be consumed by upper level AST nodes for subsequent processing in row batches, e.g., aggregations (at the same level) and upper level SQL clauses (if a sub-query is involved).

We have evaluated the scalability of ISP on both multi-core CPU and GPU equipped clusters to accelerate spatial join processing, including both filter and refinement phases. Similar to SpatialSpark, comparisons have been made with other existing works. The results and performance studies will be presented in the experiment chapter in Section 5.3.2.

4.3.3 LDE

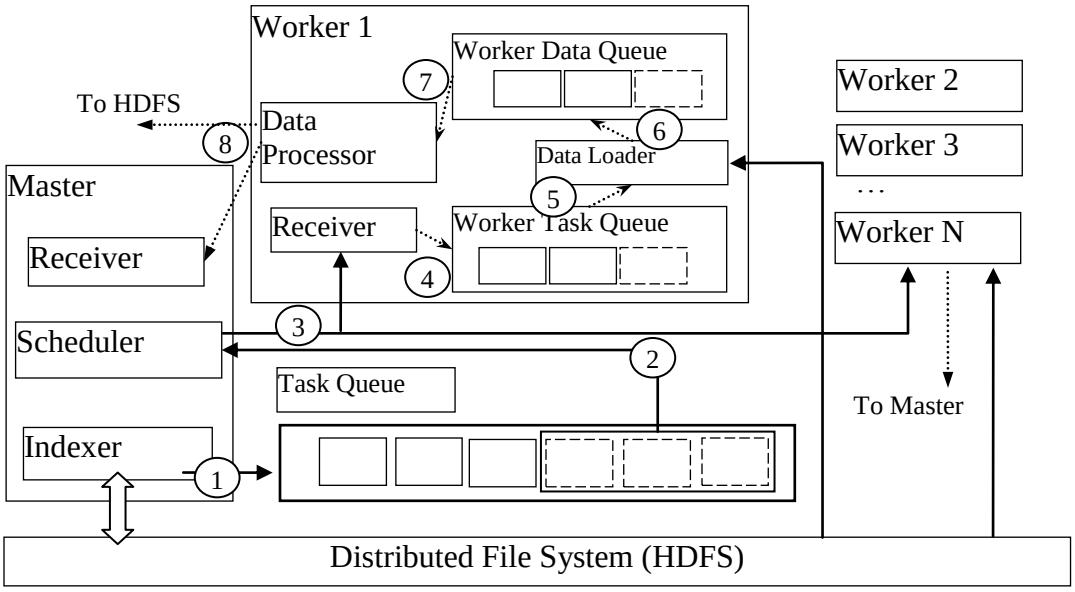


Figure 33 LDE Architecture

Based on the experiences of developing SpatialSpark and ISP, we have observed that the infrastructure overheads in distributed spatial join processing can be very expensive. Meanwhile, extending existing systems such as Impala is very challenging, because the spatial processing module is required to be tightly coupled with the underlining infrastructure in ISP. Even though Spark provides a framework that spatial extensions can be relatively easily developed, unfortunately, it is difficult to utilize hardware accelerators such as GPUs to further improve performance because of the restrictions of the underlining runtime system. As such, we have developed a lightweight distributed execution engine, namely *LDE,* to support efficient distributed large-scale spatial data processing. We design LDE by taking consideration of three key aspects. First of all, LDE is a lightweight framework that targets for domain specific applications, especially spatial data processing. Second, LDE runs on distributed environments so that large-scale datasets can be processed efficiently. Third, hardware accelerators such as

multi-core CPUs and GPUs can be integrated in order to fully exploit computing power on individual computing nodes.

The architecture of LDE is illustrated in Figure 33. As shown in the figure, our execution engine consists of a master node and a set of worker nodes. For both master and worker nodes, we use threads and queues to achieve non-blocking calls. We have adopted Apache Thrift[22] to communicate among multiple machines, including serialization, data transfer and deserialization. Data is stored in a distributed file system such as HDFS so that all workers are able to perform random disk access.

Given a particular application to be executed on LDE, based on the gathered dataset information, the indexer divides the original problem into multiple independent tasks and pushes the tasks into a task queue asynchronously (Step 1). The scheduler, which runs as a demon thread, consumes the task queue and dispatches available tasks to all workers for local processing (Step 2 and 3). The dispatch of tasks is designed to be non-blocking. On each worker node, a receiver thread is launched to accept tasks from the master node by listening to a predefined port. Received tasks are pushed into its own task queue (Step 4), where the capacity of the task queue is configured during system initialization. The worker task queue is initially filled by the master node based on the advertised capacity. Upon successfully completing a task, the worker node signals the master node to send a new task. 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. When the worker task queue is not empty, the data loader thread consumes the task queue

[22] https://thrift.apache.org/

and loads relevant data for each task from the distributed file system in the framework (Step 6). Loaded data are kept in memory and pushed into a data queue. Notice that the data loading is also designed in a non-blocking manner, so the expensive IO overhead can be minimized with asynchronous processing by a data processor thread (Step 7). Finally, the worker reports status back to the master and relevant output will be saved into the distributed file system (Step 8).

We have developed both broadcast based and partition based spatial joins using the LDE framework. In broadcast based spatial join, the large dataset is divided into equal ranges and the small dataset is broadcast to all computing nodes. In the task queue of the master node, the task is defined as a pair of a range and the broadcast dataset. When worker receives the task, it loads the broadcast dataset from the underlying distributed file system (e.g., HDFS). The broadcast dataset will be persisted in memory on each node to avoid unnecessary disk access during the process. In this design, we do not load datasets from the file system on the master node. Instead, only file locations and corresponding offsets are sent from the master node to worker nodes. By this means, substantial disk IO can be avoided on the master node, and the data loading will be delayed until worker nodes start to process. Meanwhile, loading data at each worker node can also benefit from the scheduling of distributed file system which may improve the performance of distributed IOs. On each worker node, similar to SpatialSpark and ISP, we apply our single-node parallel spatial join techniques (see Section 4.1 for details). In LDE, a spatial index (such as R-tree or grid-file) is created and kept in memory to speed up local spatial join processing.

In partition based spatial join, there are two design options in LDE. The first option is similar to SpatialSpark, where a partition schema is generated by sampling the input datasets and both datasets are repartitioned according the generated schema. Then, each partition is assigned

as an independent task in the LDE framework and processed using local parallel spatial join on a single computing node. Another option is to index both datasets before performing spatial join. The process is as follows. First, distributed indexes from both datasets are loaded in the master node. Then, a local spatial join is performed on the distributed indexes and matched pairs are assigned as tasks that will be processed in our LDE framework. Here, a matched pair refers to a pair of intersected partitions from the indexed datasets. The idea is similar to indexed distributed join developed in existing work SpatialHadoop [25]. However, our LDE framework is much simpler than Hadoop runtime and we have more control over the whole process. As such, we could potentially have better utilization of all available resources with less system overhead. Furthermore, we can benefit from in-memory processing and take advantages of state-of-the-art parallel hardware such as GPUs which are difficult when using existing JVM based systems.

4.4 Summary

We have introduced several designs for single-node parallel spatial join and multi-node distributed spatial join in this chapter. For single-node parallel spatial join, we have developed parallel designs for both filter and refinement phases. In this work, we assume the data is static or near-static, which means, the updates on the dataset is not very frequently. As a result, the spatial join techniques we have developed do not require maintaining dynamic indexing structures and indexing structures are generated via efficient bulk loading techniques. Even though our spatial join designs do not provide direct support for datasets with continuous updates, spatial joins with moderate update frequencies can be performed in a batch manner where the indexing structure can be re-generated. As the parallel bulk loading techniques we have developed are very efficient, regenerating indexing structures can be very fast. Meanwhile,

if only one side of a spatial join contains updates, lightweight indexing techniques can be applied, for example, the technique introduced in 4.1.1.

In this chapter, we have also introduced techniques to improve the refinement phase using SIMD operations which has not been well studied in the past. By comparing with existing geometry libraries, our designs are capable of taking advantages of current generation of commodity parallel hardware. We have also introduced the design of intermediate filtering that computes coarse relationship for each candidate pair as an extension to the classic filter-refinement framework for spatial join.

For very large scale datasets, they may be beyond a single node's capacity in terms of memory and computation, which requires distributed spatial join processing. Our goal is to combine both single node parallel techniques such as GPU with state-of-the-art big data platforms, which will provide another level of parallelism. First, we have developed two spatial join designs, i.e., broadcast- and partition-based methods. The two designs are targeting at different applications according to the characteristics of input datasets. The spatial partition based method is a general approach by spatially dividing datasets into partitions, and no communication is needed between partitions so that they can be processed independently. For asymmetric input datasets, we have developed a broadcast based method which sends the small dataset to all nodes instead of performing expensive spatial partitioning. As a result, expensive data reordering can be saved and significant speedup has been achieved.

In this chapter, we have introduced three prototype systems, i.e., SpatialSpark, ISP and LDE. These three implementations are built based on different platforms and representing

different needs of real world applications. For applications that are more concerned about compatibility and extendibility, SpatialSpark will be the choice even though it cannot effectively utilize hardware accelerators. LDE is a specialized implementation targeting on specific applications where performance is most crucial, as LDE can take advantages of existing parallel hardware and has least system overhead among the three systems. However, LDE is developed from scratch and robustness and usability are under active improvements. ISP is between SpatialSpark and LDE, which has both compatibility and efficiency. However, the development complexity and low extendibility of Impala limit its practical applicability to processing spatial data.

In summary, we have developed designs for single-node parallel spatial join and distributed spatial join. Evaluations and performance studies of the three prototype systems using real world datasets will be presented in Chapter 5.

Chapter 5 Evaluation and Performance Study

To justify the feasibility and effectiveness of our designs introduced in the previous chapters, we have conducted evaluations and performance studies using both benchmark datasets and datasets from read world applications. In this chapter, we first present performance evaluations on single node techniques, including data-parallel R-tree and grid-file indexing, using both multi-core CPUs and GPUs. In the second part of this chapter, we conduct performance study on distributed designs for multi-node prototype systems, i.e., SpatialSpark, ISP and LDE, including performance comparison with SpatialHadoop and HadoopGIS.

Table 3 Machine Specifications

Name	Hardware	Software
WS-1	A workstation with two Intel E5405 processors at 2.0 GHz (8 cores in total) and an NVIDIA Quadro 6000 GPU with CUDA 5.0	Ubuntu-10.04, GCC 4.6.3, Intel TBB 2.2, Thrust 1.6
WS-2	A workstation that has dual 8 core CPUs at 2.6 GHz, 128 GB memory, 8 TB HDD and NVIDIA GTX Titan GPU with 6 GB graphics memory and 2,668 cores.	CentOS 6.5, Hadoop 2.5.0 from Cloudera CDH 5.2.0, GCC 4.9.0, Intel TBB 2.2, Thrust 1.6
EC2	A cluster is built using Amazon EC2 instances (*g2.2xlarge*), each instance is 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.	CentOS 6.5, Hadoop 2.5.0 from Cloudera CDH 5.2.0

5.1 Setup

For experiments running on a single node, we have prepared two workstations equipped with multi-core CPUs and GPUs, and their specifications are listed as WS-1 and WS-2 in Table 3. For experiments running on multiple nodes, we have prepared two clusters, one uses a single node (WS-2), and the other is based on Amazon EC2 instances. The hardware and software specifications are also listed in Table 3.

For data-parallel R-tree evaluation, we have adopted a benchmark dataset from [1], which is designed to evaluate R-tree indexing. The specifications of the benchmark are listed at the top of Table 4 (*abs02, dia02, par02,* and *rea02*) and the related queries are shown in Table 5. For parallel spatial join evaluations including single-node and multi-node techniques, we have prepared two datasets related to New York City taxi trip analysis, which is a real world point-in-polygon test based spatial join application. The first dataset (*taxi*) has approximately 170 million pickup locations in 2009 from New York City taxi trip data, which are in the format of latitude and longitude. The other dataset (*nycb*) is a polygon dataset which is derived from NYC Census 2000 dataset[23]. The *nycb* dataset has about 40 thousand census block polygons with more than 5 million vertices. Aligning GPS locations to a street network is also widely used in taxi trip analysis, which can be represented as nearest neighbor search based spatial join. As such, we have derived a dataset (*lion*) from NYC street network (LION[24]) dataset, which has about 150 thousand polylines. In addition to the NYC taxi trip analysis, we have also prepared another point-in-polygon test based spatial join application, which is joining species occurrence records

[23] http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml

[24] http://www.nyc.gov/html/dcp/html/bytes/dwnlion.shtml

of Global Biodiversity Information Facility (GBIF) repository (snapshot 08/02/2012, termed as *gbif*) with a polygon dataset from World Wild Fund (WWF) global ecological region data (termed as *wwf*). Different from the taxi trip analysis in which polygons are usually small, global ecological regions are usually very large and require expensive geometry computation. In this chapter, there are some experiments on certain systems and configurations fail to run on the full datasets, so we also generate two sampled *gbif* datasets called *G10M* and *G50M* which contain 10,000,000 and 50,000,000 points, respectively.

For performance comparison with SpatialHadoop and HadoopGIS, we adopt datasets provided by SpatialHadoop data portal[25], namely *edges* and *linearwater*. We have also derived three sampled datasets because not all experiments can run on the full datasets. The three sampled datasets include 1 month data from the full taxi dataset (referred as *taxi1m*) and 10% sample of the TIGER datasets (*linearwater0.1* and *edges0.1*). All datasets that have been used in this chapter are listed in Table 4.

[25] http://spatialhadoop.cs.umn.edu/datasets.html

Table 4 Datasets Sizes

	Dataset	Type	# of Records	Related Sections
Benchmark	*abs02*	MBR	1000000	5.2.1(R-tree)
	dia02	MBR	1000000	5.2.1(R-tree)
	par02	MBR	1048576	5.2.1(R-tree)
	rea02	MBR	1888012	5.2.1(R-tree)
Real world	*taxi*	Point	169,720,892	5.2.2(Grid-file), 5.3.1(SpatialSpark), 5.3.2(ISP), 5.3.3(LDE)
	taxi1m	Point	2,267,313	5.3.1(SpatialSpark), 5.3.2(ISP)
	nycb	Polygon	38,839	5.2.2(Grid-file), 5.3.1(SpatialSpark), 5.3.2(ISP), 5.3.3(LDE)
	lion	Polyline	147,012	5.3.1(SpatialSpark)
	gbif	Point	375,171,681	5.3.1(SpatialSpark), 5.3.2(ISP), 5.3.3(LDE)
	wwf	Polygon	14,485	5.3.1(SpatialSpark), 5.3.2(ISP), 5.3.3(LDE)
	G10M	Point	10,000,000	5.3.1(SpatialSpark), 5.3.2(ISP)
	G50M	Point	50,000,000	5.3.2(ISP), 5.3.3(LDE)
	linearwater	Polyline	5,857,442	5.3.1(SpatialSpark), 5.3.3(LDE)
	edges	Polyline	72,729,686	5.3.1(SpatialSpark), 5.3.3(LDE)
	linearwater0.1	Polyline	585,809	5.3.1(SpatialSpark)
	edges0.1	Polyline	7,271,983	5.3.1(SpatialSpark)

Table 5 Specs of Queries

	Query size	Min # of answers	Max # of answers	Avg # of answers
abs02-Q1	1,000,000	1	1	1
abs02-Q2	10,000	50	150	99.8
abs02-Q3	3,164	500	1,500	992
dia02-Q1	1,000,000	1	4	1.26
dia02-Q2	10,000	50	150	99.8
dia02-Q3	3,164	500	1,500	992
par02-Q1	1,048,576	1	10	2.11
par02-Q2	10,485	50	150	99.8
par02-Q3	3,318	500	1,500	992
rea02-Q1	1,888,012	1	9	1.2
rea02-Q2	18,880	50	162	101
rea02-Q3	5,974	501	1,514	999

5.2 Parallel Spatial Data Management on Single-Node

5.2.1 Data-Parallel R-tree Implementation

We have implemented data-parallel R-tree using parallel primitive library for both tree construction and batch query. Both the multi-core CPU and GPU parallel versions are developed for comparison purpose. We have evaluated our implementations using both WS-1 and WS-2, which represent two different generations of commodity parallel hardware.

The major component in R-tree construction that dominates the overall performance is the sorting phase (Section 3.2.3.2). We have used sort implementations in existing libraries such as STL, TBB and Thrust. In this set of experiments, we empirically set R-tree fan-out to 4 and use *x*-coordinates of MBR centroids as sorting keys. The experiment results are given as Figure 34**A** (using WS-1) and Figure 34**C** (using WS-2), where "*CPU-serial*" denotes CPU serial implementation, "*CPU-parallel*" denotes the CPU parallel implementation, and, "*GPU-primitive*" denotes the GPU implementation based on parallel primitives.

From Figure 34A we can observe that, when datasets are relatively small, parallel CPU implementations outperform GPU implementations. One explanation is that GPU parallel processing power is not fully exploited for small datasets and the overheads of utilizing parallel library cannot be hidden. We also observe that the runtimes for GPU implementations increase much slower than those of parallel CPU implementations which might indicate better scalability of the GPU implementations. In particular, when datasets become large enough that can hide library overheads, GPU implementations are several times faster than parallel CPU implementations. Following this trend, we might be able to predict that GPUs are capable of

achieving better performance when bulk loading larger datasets. However, we should be aware that GPU memory capacities are usually limited when compared with CPU memory capacities. Therefore large datasets might not be able to completely reside in GPU memory. In this case, however, we still can process such large dataset using data partition techniques which are left for future work.

Comparing Figure 34C with Figure 34A, we can observe that the runtimes of both CPU and GPU are lower on WS-2, this is because the hardware on WS-2 is newer and more powerful than WS-1. Although the GPU of WS-2 has more cores than WS-1, the performance only improves 20% which does not achieve the level of speedup as one might expect. By breaking down the runtimes, we find that the sorting phase on WS-2 is 2X faster than that of WS-1. However, the tree construction phase does not improve, which is primarily due to underutilization of hardware resource. As such, the overall improvement for the newer GPU is limited. On the other hand, the runtimes on WS-2 are about 2.7X lower comparing with WS-1 on multi-core CPUs because of more powerful CPUs equipped on WS-2 (such as more cores, higher frequency, larger cache size, etc.). This explains that the absolute speedups for GPU over CPUs on WS-2 are lower than those on WS-1.

We have also implemented and evaluated the STR R-tree bulk loading algorithm (Section 3.2.3.2) on multi-core CPUs and GPUs and experiment them on both WS-1 and WS-2. The results are given in the right chart of Figure 34 (**B** for WS-1 and **D** for WS-2) where "*STR-CPU-Parallel*" denotes the multi-core CPU implementation and "*STR-GPU*" denotes the GPU implementation. From the results, our GPU implementation has achieved about 4X speedup over

the multi-core CPU implementation on WS-1 and about 3X speedup on WS-2. Similar to low-*x*, the speedup on WS-2 is lower because it has more powerful CPUs. Based on the results shown in Figure 34, low-*x* bulk loading is faster than STR bulk loading for both CPU and GPU implementations. The STR R-tree bulk loading, as discussed in Section 3.2.3, requires multiple sorts at each level. Thus, as expected, the overall performance of the STR R-tree bulk loading technique is not as fast as the low-*x* bulk loading technique that only sorts once. However, from our query benchmark results, R-tree generated by the STR bulk loading technique usually has better quality comparing with low-*x* bulk loading and results in faster query processing, a feature that is desirable.

We also compare the performance of batch query processing on GPUs with multi-core CPUs. The multi-core CPU implementations utilize all available cores in the system using OpenMP where each core is responsible for a single query. As shown in Figure 35, our GPU implementations have achieved about 10X speedup on average when compared with multi-core CPU implementations for WS-1. For WS-2, the speedup is about 3X because more CPU cores are used as we discussed previously. For queries labeled with Q1 which use small query windows, GPU implementations do not show advantages over multi-core CPU implementations. However, as the size of query results in each query window increases, GPU based implementations outperform their counterparts significantly.

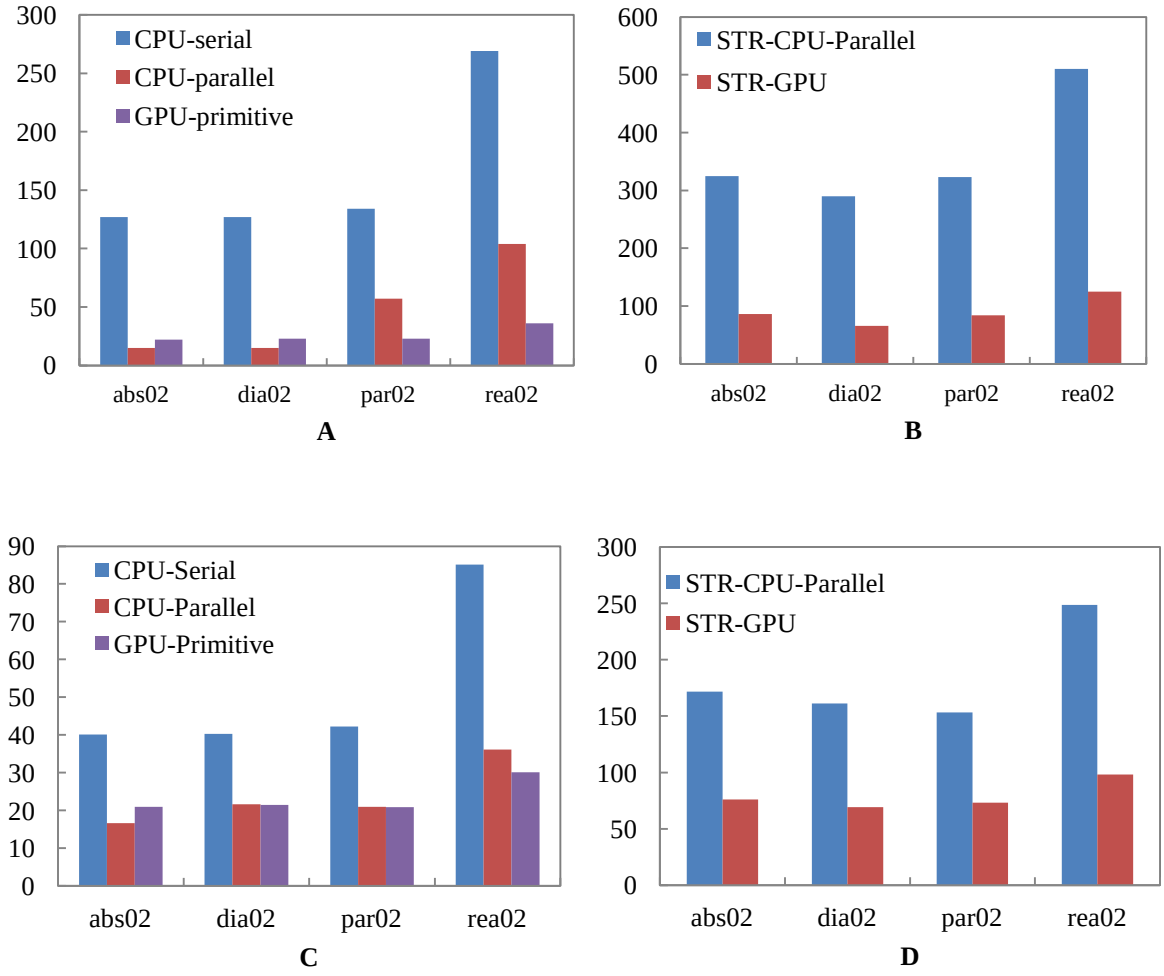


Figure 34 Performance of R-tree Construction (time in milliseconds)

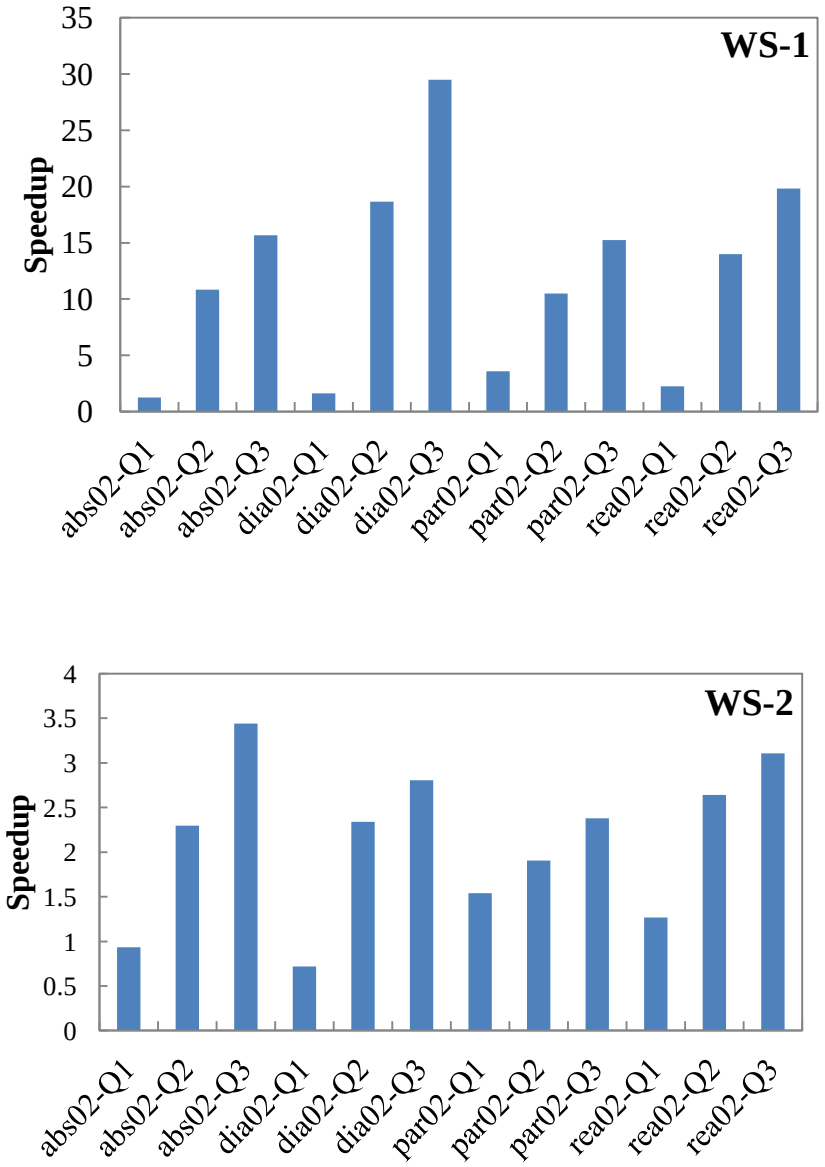


Figure 35 Speedups of GPU-based Implementations over Multi-Core CPU-based Implementations for Spatial Window Query Processing

5.2.2 Grid-file based Spatial Join

We have implemented grid-file based spatial join on both multi-core CPUs and GPUs, which is used to demonstrate the effectiveness of grid-file indexing and single node parallel spatial join

designs. The design of grid-file based filtering uses the batch query processing technique that has been introduced in Section 3.2.2. We implement a point-in-polygon test based spatial join on GPUs using the grid-file based parallel filtering. In this application, point quadrants are generated first using the Quadtree index as introduced in Section 3.2.4 and only MBRs of point quadrants are used for the filter phase. For the refinement phase, each GPU block is responsible for processing a matched pair of point quadrant and a polygon. Within a GPU block, each thread is assigned to process a point for the point-in-polygon test using the classic ray-casting algorithm [82].

For comparison purposes, we have implemented the same spatial join using open source GIS packages, i.e., libspatialindex[26] to index polygon data using R-tree, and, GDAL[27], which implicitly uses GEOS, to perform point-in-polygon test. The CPU implementation assigns each point to a thread which performs query against the indexed polygons. If the point falls within any of the bounding boxes of polygons, the polygon identifiers will be returned for the subsequent refinement phase. It is clear that, while the polygons do not spatially overlap, their bounding boxes can overlap and a point query may return multiple polygons for point-in-polygon test in the refinement phase. The CPU implementation performs the point-in-polygon test for each of the polygons in the query result set and breaks if any of the test returns true. The performance of our GPU implementation is an end-to-end runtime of 11.2 seconds on WS-1 and 7.7 seconds on WS-2. In contrast, the serial CPU implementation takes 54,819 seconds (15.2 hours) on WS-1. As such, a significant speedup of 4,910X has been achieved. Note that we have not included the

[26] http://libspatialindex.github.com/

[27] http://www.gdal.org/

disk I/O times to load the points and polygons as this is one-time cost and is not directly related to the spatial join. Furthermore, as discussed before, these data are stored as binary files on disk. With a sustainable disk I/O speed of 100 MB per second, the point and polygon data can be streamed into CPU main memory in about 15 seconds. Since the disk I/O time is comparable to the spatial join time, even if the disk I/O times are included, the order of speedup will not be changed.

We attribute the 3-4 orders of improvements to the following factors. First, all the points, polygons and auxiliary data are memory resident in our GPU implementation. In contrast, the open source GIS packages are designed to be disk resident and data and indexes are brought to CPU memory dynamically. While the sophisticated design is necessary for old generations of hardware with very limited CPU memory, current commodity computers typically have tens of gigabytes of CPU memory which renders the sophisticated design inefficient and unnecessary. We also have observed that the open source packages use dynamic memory and pointers extensively which can result in significant cache and TLB[28] misses. Second, in our GPU implementation, we have divided points into quadrants before we query against the polygons in the filter phase using a GPU based grid file indexing structure. In the serial CPU implementation, each point queries against the polygon dataset individually. While the polygon dataset is indexed, each point query needs to traverse from the root of the R-tree of the polygon dataset to leaf nodes, which is quite costly. Third, in addition to the improved floating point computation on GPUs, the massively data parallel GPU computing power is utilized for all phases of the

[28] https://en.wikipedia.org/wiki/Translation_lookaside_buffer

spatial join process, including generating point quadrants, filtering quadrant-polygon pairs and performing point-in-polygon test in computing blocks.

5.3 Parallel Spatial Data Management on Multi-Node

5.3.1 SpatialSpark

We have implemented SpatialSpark for both broadcast based and spatial partition based spatial joins introduced in Section 4.2. In our preliminary implementation, JTS library is used for spatial indexing (R-tree) and geometry operations. We have evaluated SpatialSpark for two spatial join operations, including point-in-polygon test based spatial join and nearest-neighbor-search based spatial join. In the point-in-polygon test based spatial join, we use *taxi* and *nycb* datasets. For the nearest-neighbor-search based spatial join, we use *taxi* and *lion* datasets. All datasets are formatted and stored as text files in HDFS with geometries (points, polylines and polygons) represented in WKT format. In addition to the taxi point dataset, we also use the GBIF species occurrence data (*gbif*) joining with *wwf* dataset. In this experiment, we only use *G10M* because using the full dataset (*gbif*) takes too long to finish.

We have evaluated the performance of the four experiments on a 10-node Amazon EC2 cluster (see Table 3 for specifications) and the results are plotted in Figure 36. For *taxi-lion* experiments, we empirically use 100 feet and 500 feet as search radius. We have also varied instance numbers for scalability tests in the four experiments. We are not able to use fewer than 4 nodes for the experiments due to the memory limitation of the EC2 instances (15 GB per node). In Figure 36, all four experiments scale linearly when the number of instances increases. As such, SpatialSpark achieves very good scalability.

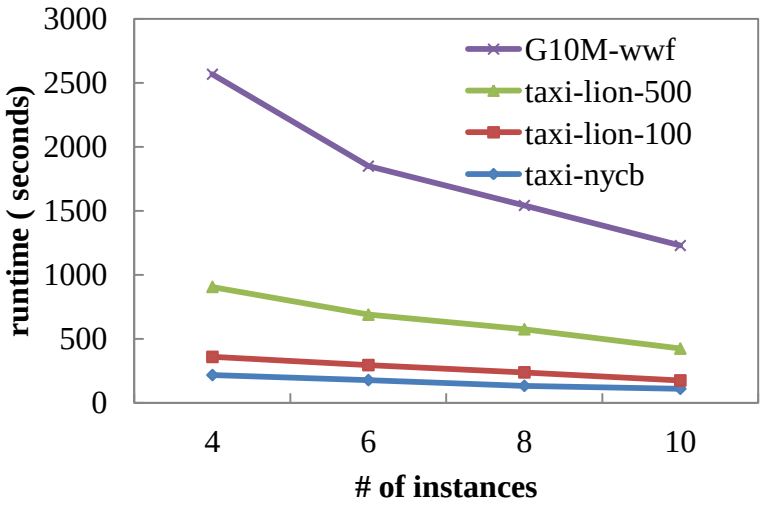


Figure 36 SpatialSpark Performance

In order to demonstrate the efficiency of SpatialSpark, we have also conducted performance comparison with existing works, i.e., Hadoop-GIS and SpatialHadoop. The first experiment is designed to evaluate point-in-polygon test based spatial join, which uses *taxi* and *nycb*. The second experiment is designed to evaluate polyline-with-polyline intersection based spatial join using *edges* and *linearwater*. In addition to the full datasets, sampled datasets, i.e., *taxi1m, linearwater0.1* and *edges 0.1*, are also used because not all experiment configurations are successful on the full datasets. The performance of the sample datasets can provide an idea of the relative performance among the three prototype systems when one or more systems cannot handle the full datasets successfully. In the next two subsections, we will present evaluation results on both the full datasets and the sampled datasets.

5.3.1.1 Results Using Full Datasets

The end-to-end runtimes (in seconds) for the two experiments (*taxi-nycb* and *edge-linearwater*) under the four configurations on the three systems are listed in Table 6. The reported runtimes include indexing the two input datasets and performing the distributed join,

i.e., end-to-end runtimes. It can be seen that *HadoopGIS* fail on all the experiments using the full datasets; *SpatialHadoop* is successful in all the experiments while *SpatialSpark* is in between. The top reason for *HadoopGIS* to fail is broken pipeline, which is typical in Hadoop Streaming when the data that pipes through multiple processes is too big. The primary reason for *SpatialSpark* to fail is out of memory due to Java heap size issue, which is expected to be solved in the future releases of Spark. While *SpatialSpark* is successful for both the workstation and EC2-10 configurations, it failed under EC2-8 and EC2-6 configurations. We note the workstation has 128 GB memory and the aggregated memory capacity of the EC2-10 cluster is 150 GB, which are sufficient for *SpatialSpark* to experiment on the full datasets. We also expect the new release of Spark can handle the problem by taking advantages of external disk storage.

Table 6 End-to-End Runtimes of Experiment Results of Full Datasets (in seconds)

		WS-2	**EC2-10**	**EC2-8**	**EC2-6**
taxi-nycb	HadoopGIS	-	-	-	-
	SpatialHadoop	3,327	2,361	2,472	3,349
	SpatialSpark	3,098	813	-	-
edge-linearwater	HadoopGIS	-	-	-	-
	SpatialHadoop	14,135	5,695	8,043	9,678
	SpatialSpark	4,481	1,119	-	-

When the available memory capacity is sufficient, it can be seen from Table 6 that *SpatialSpark* is significantly faster than *SpatialHadoop*. Under EC2-10 configuration, *SpatialSpark* is 2.9X and 5.1X faster than *SpatialHadoop* for the two experiments, respectively. The results are different under the workstation configuration where *SpatialSpark* is 3.2X faster for the *edge-linearwater* experiment but is only 1.07X faster for the *taxi-nycb* experiment. A possible explanation is that the *taxi-nycb* experiment is much more disk I/O intensive than the

edge-linearwater experiment and the performance of the workstation is significantly limited by its single-node disk I/O bandwidth. When disk I/O is not a limiting factor (either by using distributed I/O or the experiment is more computing bound as in the *edge-linearwater* experiment), the speedups of *SpatialSpark* over *SpatialHadoop* have clearly demonstrated the efficiency of in-memory processing.

5.3.1.2 Results Using Sampled Datasets

The runtimes of the *taxi1m-nycb* and *edge0.1-linearwater0.1* experiments are listed in Table 7. Since the performance of the three EC2 configurations are roughly the same for all the three systems (which may indicate poor scalability), we only show the results under the workstation and EC2-10 configurations. We list the breakdown runtimes to provide a better idea on the runtime distributions: column IA is the runtime for indexing the left side input dataset (*taxi1m* and *edge0.1*), column IB is the runtime for indexing the right side input dataset (*nycb* and *linearwater0.1*), column DJ is the runtime for distributed spatial join, and, column TOT is the summation of the three.

Table 7 Breakdown Runtimes of Experiment Results Using Sample Datasets (in seconds)

		WS-2				**EC2-10**			
		IA	IB	DJ	TOT	IA	IB	DJ	TOT
taxi1m-nycb	HadoopGIS	206	54	3,273	3,533				-
	SpatialHadoop	227	52	230	482	647	187	183	1,017
	SpatialSpark				216				67
edge0.1-linearwater0.1	HadoopGIS	1,550	488	1,249	3,287				-
	SpatialHadoop	1,013	307	220	1,540	756	596	106	1,458
	SpatialSpark				765				48

Although *HadoopGIS* still fail under the EC2-10 configuration for both experiments, it is successful under the workstation configuration. This makes it possible to compare its

performance with *SpatialHadoop* and *SpatialSpark* directly. The runtimes for *SpatialSpark* are end-to-end times as it is difficult to measure each individual step due to the asynchronous data communication/computation in Spark. The results listed in Table 7 suggest that, while the indexing times are comparable in both *HadoopGIS* and *SpatialHadoop*, *SpatialHadoop* is 14X and 5.7X faster than *HadoopGIS* for distributed joins (as reported in the DJ column) in the two experiments, respectively. While excessive disk I/O and string parsing might be important factors in contributing to the low performance of *HadoopGIS*, our in-house experiments have identified that the C++ based GEOS geometry library used in *HadoopGIS* can be several time slower than the Java-based geometry library (i.e., JTS) used in *SpatialHadoop* and *SpatialSpark*, which might be another major factor. We thus exclude *HadoopGIS* from further comparisons.

When comparing the end-to-end runtimes between *SpatialHadoop* and *SpatialSpark* using the sampled datasets, *SpatialSpark* is about 2.2X faster under the workstation configuration but is about 15X faster under the EC2-10 configuration for the *taxi1m-nycb* experiment. Similar results, i.e., 2.0X and 30X under the EC2-10 configuration, can be observed in the *edge0.1-linearwater0.1* experiment. The result exceeds our expectation when compared with the speedups using the full datasets. A careful investigation reveals that indexing times under the EC2-10 configuration dominates in both experiments using the sampled datasets. These are quite different from the full dataset experiment results that distributed join (DJ) consumes most of the runtime, which are 1,950s out of 3,327s for *taxi-nycb* experiment under workstation configuration, 1,282s out of 2,361s for *taxi-nycb* experiment under EC2-10 configuration, 9,887s out of 14,135s for *edge-linearwater* under workstation configuration and 3,886s out of 5,695s for *edge-linearwater* under EC2-10 configuration. An explanation is that, indexing under EC2-10

configuration involves significant data shuffling among the 10 distributed computing nodes which can be very expensive for *SpatialHadoop*. In contrast, distributed joins under the EC2-10 configuration can be significantly sped up by distributed I/Os in *SpatialSpark*.

When comparing the distributed join times (DJ) only, *SpatialHadoop* takes only 220s in *edge0.1-linearwater0.1* experiment under the workstation configuration, which is significantly lower than the indexing runtimes. This may indicate the Hadoop infrastructure overheads for small datasets on a single computing node may be high. We note that the end-to-end runtime of *SpatialSpark* (765s) is much larger than the distributed join (DJ) runtime but it is only half of the total (TOT) runtime of *SpatialHadoop*. Under EC2-10 configuration, *SpatialSpark* is 2.7X and 2.2X faster than *SpatialHadoop* with respect to distributed join (DJ) runtimes for the two experiments, respectively. The results are consistent with the experiments using the full datasets, which are 1.8X (1282/712) and 3.5X (3886/1119) for the two experiments under EC2-10 configuration. It is clear that the speedups of *SpatialSpark* over *SpatialHadoop* are mostly due to the ability to reduce unnecessary disk accesses by pipelining the process completely in memory as the underlying algorithms are the same and they use a same geometry library (JTS).

5.3.2 ISP

We have conducted performance evaluation on two sets of experiments for ISP. The first experiment is performed using *taxi* and *nycb*. The other experiment uses *gbif* and *wwf*, which shows performance on complex polygons. We first report the performance of ISP-MC and ISP-GPU on WS-2 and then report the performance of the standalone versions of the two prototypes

on the same machine to understand system overhead. Finally, the performance results on EC2 clusters are reported for discussions on scalability.

The single-node performance for the two experiments is listed in the first two columns of Table 8. The runtimes are 96 seconds for *taxi-nycb* and 1,822 seconds for *gbif-wwf* for the ISP-GPU implementation. ISP-MC performs slower than ISP-GPU but is still comparable: 130 seconds for *taxi-nycb* and 2,816 seconds for *gbif-wwf*. ISP-GPU is 1.35X (130/96) faster than ISP-MC for *taxi-nycb* and 1.55X (2816/1822) faster than ISP-MC for *gbif-wwf*. The comparable performance between ISP-GPU and ISP-MC is largely due to applying the same set of data parallel deigns and parallel primitives based implementations, which are efficient on not only GPUs but also multi-core CPUs. Similar to the experiment reported in the previous section, the serial implementation using libspatialindex and can only achieve 138 points per second using a subset of GBIF data with 10 million points on a single CPU core. In contrast, ISP-GPU has achieved a rate of 206 thousand points per second using a single GPU which amounts to a 1,491X speedup. When comparing ISP-MC with the baseline implementation (965X speedup), while the multiple CPU cores and higher CPU frequency may explain up to 21X speedups (16*2.6/2.0), the rest of the speedups are largely due to our data parallel designs and better use of memory capacity.

We have also implemented two standalone versions without Impala and run them on the same workstation. The results are listed in the last two columns of Table 8. Clearly, the system infrastructure overhead is quite significant for ISP-GPU: almost 50% (46s) in the *taxi-nycb* experiment and 17% (324s) in the *gbif-wwf* experiment. The overheads are 20% and 8.3% for

ISP-MC, respectively. Although still significant, the infrastructure overheads are much smaller for ISP-MC than for ISP-GPU in both experiments. As the experiments become more floating point computing intensive where computation becomes dominate, we expect the system infrastructure overheads continue to decline for both ISP-GPU and ISP-MC.

Table 8 ISP Performance on Single Node

	ISP-GPU	**ISP-MC**	**GPU-Standalone**	**MC-Standalone**
taxi-nycb (s)	96	130	50	89
GBIF-WWF(s)	1822	2816	1498	2664

We have also conducted scalability tests on Amazon EC2 clusters with up to 10 instances. As the memory capacity of the instances is 15 GB, we are not able to run the *taxi-nycb* workload with four or fewer nodes. Also due to the memory capacity constraint, we are not able to experiment on the complete WWF dataset on the 10-node cluster. As such, we use the sampled dataset *G50M* and label the experiment as *G50M-wwf*. The scalability results for *taxi-nycb* and *G50M-wwf* experiments are plotted in Figure 37. For the *taxi-nycb* experiment, as the number of computing nodes increases, the runtime decreases almost linearly that indicates good scalability for both GPU and CPU implementations. For the *G50M-wwf* experiment, the scalability of ISP-GPU is approximately linear until the number of nodes is increased to above 8. Almost no performance gains are observed when the number of instances is increased from 8 to 10. On the other hand, ISP-MC scales up to 10 nodes, although the slope is flatter when the number of instances is increased from 6 to 10 than from 2 to 6 (i.e., scalability becomes lower). Overall, there is a 1.76X speedup for ISP-MC and 1.56X speedup for ISP-GPU when the number of nodes is increased from 6 to 10 (1.66X) for the *taxi-nycb* experiment, which is very good. In

the *G50M-wwf* experiment, the speedups are 3.19X for ISP-MC and 2.57X for ISP-GPU when the number of node is increased from 2 to 10 (5X), which is still decent with respect to parallelization efficiency (defined as the ratio of performance speedup over increase of parallel processing units).

The lower speedups when the numbers of computing nodes become higher in the *G50M-wwf* experiment might be largely due to the static scheduling policy imposed by Impala. By examining the *G50M* point dataset in HDFS, we found that there were 14 HDFS data blocks, which makes the end-to-end runtime about the same using 8-13 computing nodes, as it is determined by the runtime of the computing nodes that process the most (two) blocks. Increasing the number of blocks is likely to reduce load unbalancing to scale further. However, as discussed earlier, as per-node work load decreases, GPUs will likely be underutilized and will negatively hurt the overall performance. The small per-node workload on GPUs is also likely to incur load unbalancing among GPU threads and thread blocks which may further decrease ISP-GPU performance. Since the number of CPU cores is much smaller than the number of GPU cores, the intra-node load unbalancing is less likely to be an issue for ISP-MC, which might explain its better scalability than ISP-GPU in both experiments. When comparing ISP-GPU with ISP-MC on the EC2 cluster, ISP-GPU is 1.43X to 1.63X faster for the *taxi-nycb* experiment and 2.74X to 3.24X faster for the *G50M-wwf* experiment, which are higher than the results on the workstation. This is likely due to the fact that the CPUs equipped with WS-2 have 2X cores than those on EC2 nodes while the differences among their GPUs are smaller (1.75X more CUDA cores and 1.5X GPU memory). The results may suggest that GPU acceleration is more profitable for computing nodes with less powerful CPUs.

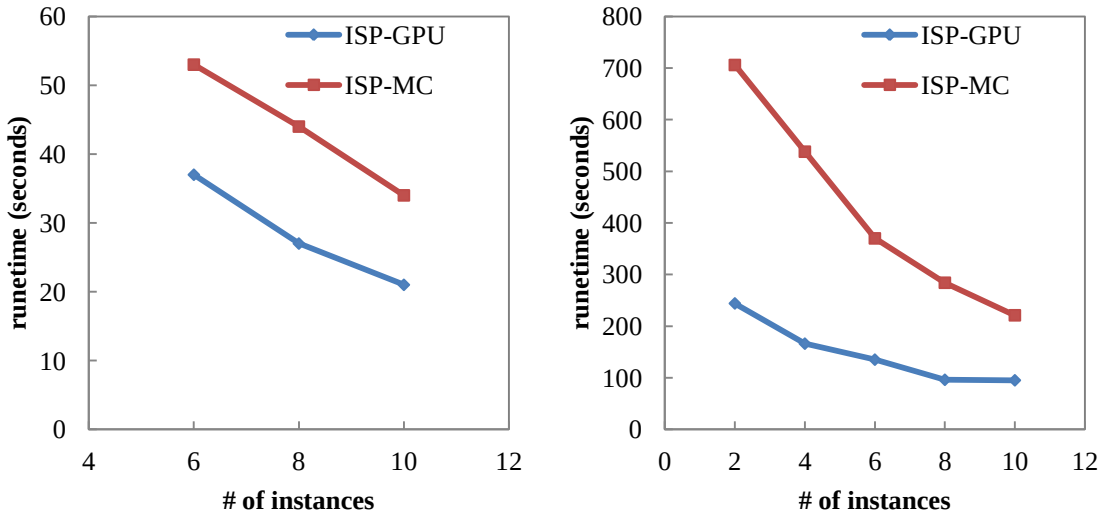


Figure 37 Scalability Test Results of ISP-GPU and ISP-MC for taxi-nycb (left) and G50M-wwf (right) Experiments

5.3.3 LDE

We use real world datasets to demonstrate the feasibility and efficiency of the distributed point-in-polygon test based spatial join technique on top of the lightweight distributed execution engine (LDE). To demonstrate advantages of LDE, we use the datasets with complex polygons, i.e., *G50M* for points and *wwf* for polygons. The same datasets have been used in ISP-based experiments in the previous subsection. 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 indexes 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 and high performance of the application with respect to the end-to-end runtime.

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 WS-2 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. All the performance results are measured in seconds and compared with ISP when appropriate.

The standalone performance and the single-node performance for the two experiments are listed in Table 9. Note that ISP and LDE have the same runtime when they run in the standalone mode, which are 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 on multi-core CPUs. As the infrastructure overheads are typically difficult to scale up (intra-node), the result agrees with the Amdahl's law well [38].

Table 9 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		380	7.89%
	LDE-MC	350	377	7.16%
GPU	ISP-GPU		241	27.80%
	LDE-GPU	174	221	21.27%

The scalability results using 2-10 Amazon EC2 nodes are plotted in Figure 38. We have avoided reporting the performance on a single node as it requires at least two nodes to count network communication overheads. 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 multi-core 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),

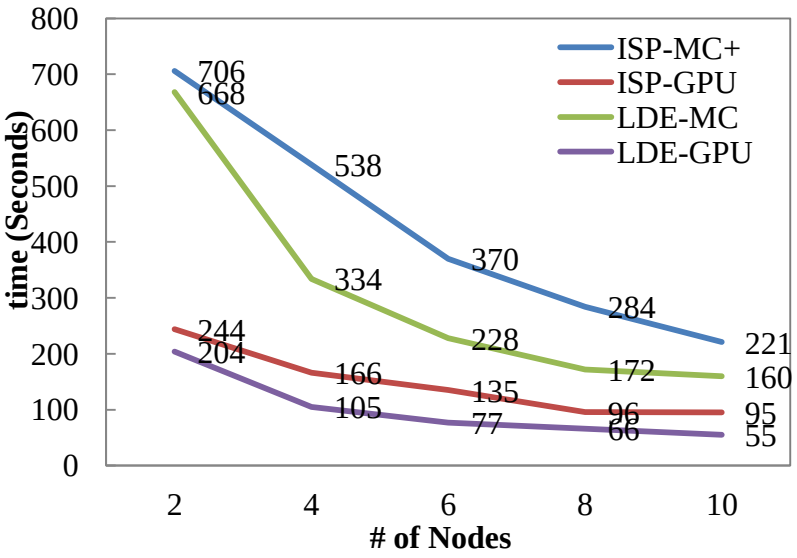


Figure 38 Scalability Comparisons between ISP and LDE on Multi-core CPU and GPU Equipped Clusters

LDE is able to further achieve 1.20X (66/50) speedup, which is impressive. As a summary, LDE has achieved both higher efficiency and higher scalability on both multi-core CPUs and GPUs when compared with the ISP implementations.

In addition to broadcast based spatial join, we have also evaluated spatial partition based spatial join implementation using the LDE framework. In this set of evaluation, we use two additional large datasets, *edges* and *linearwater*. The sizes of the two datasets are 23.8 GB and 8.4 GB respectively. Since both datasets are large, the broadcast based spatial join cannot be applied because neither can be broadcast and resident in memory. For comparison purpose, we also include runtimes of SpatialHadoop using the same set of workloads. The end-to-end runtimes (in seconds) for the two experiments (*taxi-nycb* and *edge-linearwater*) are listed in Table 10. The *taxi-nycb* experiment performs point-in-polygon test based spatial join and the *edge-linearwater* experiment performs polyline intersection base spatial join. Comparing with

SpatialHadoop, the LDE implementations on both multi-core CPUs and GPU are at least an order of magnitude faster for all configurations.

The improvements come in two folds. First, the LDE framework based on C++ is much faster and lighter than general purpose JVM based frameworks such as Hadoop. The in-memory processing of LDE is also an important factor where Hadoop is mainly a disk-based system. With in-memory processing, intermediate results will not be written to disks. Second, the dedicated local parallel spatial join module can fully exploit computing power of individual computing nodes. Our data-parallel designs in the module, including both spatial filter and refinement phases, can effectively utilize current generation of parallel hardware, i.e., multi-core CPUs and GPUs. Based on the EC2 results, we could observe that decent scalability is achieved from 6-node to 10-node. When replacing multi-core CPUs with GPUs, the performance can be further improved, especially on EC2 instances where 2X speedup is achieved.

Table 10 Partition-based Spatial Join Results (end-to-end, time in seconds)

		Workstation	**EC2-10**	**EC2-8**	**EC2-6**
taxi-nycb	SpatialHadoop[29]	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

[29] spatial join time only, excluding indexing time for the two input datasets

Chapter 6 Conclusions and Future Work

6.1 Summary of Contribution

This dissertation work identifies current challenges of large-scale spatial data management, especially on how to accelerate large-scale spatial data processing on state-of-the-art parallel and distributed platforms. Data-parallel designs of spatial indexing techniques have been developed in this work, and the implementations and experimental studies reveal the performance impacts of utilizing hardware accelerators, i.e., multi-core CPUs and GPUs. As spatial join operations are crucial in many real world applications, this dissertation work develops efficient hardware accelerated spatial join designs to fully exploit computing power of a single node. To address the practical needs of the Big Data challenge, distributed spatial join has been studied in this work. The optimized single-node parallel spatial indexing and spatial join techniques are scaled out to multi-node environments that are capable of processing spatial data beyond the capacity of a single node. This dissertation work successfully integrates Big Data technologies with current generation of hardware accelerators (e.g., GPUs) for large-scale spatial data processing. Prototype systems developed in this dissertation work have demonstrated performance advantages against existing designs and implementations, which can address practical needs of large-scale spatial data management.

6.2 Discussions and Future Work

6.2.1 Spatial Indexing Techniques

We have developed three major spatial indexing techniques for the current generation of parallel hardware, especially on GPUs. Grid-file and Quadtree indexing techniques can be categorized as space-oriented indexing, where the space is decomposed and indexed. This type of indexing techniques suffers from large duplicates for objects on or near the partition boundaries. The duplication incurs significant memory pressure, which may limit the indexing structure on memory constraint systems, such as GPUs. Meanwhile, finding the optimal resolution parameter and maximum decomposition level (for Quadtree) is also challenging. Choosing the resolution parameter can be considered as a tradeoff between indexing quality and memory utilization. For Grid-file indexing, the simple indexing structure makes it attractive for developing data-parallel designs. In addition to its simplicity, it is also light-weight and effective. On the contrary, R-tree indexing is categorized as data-oriented, which means the indexing structure relies on the data rather than the space to be indexed. This makes R-Tree indexing both scalable and portable, and it does not require tuning resolution parameters. Meanwhile, objects will not overlap with partition boundaries so they are not duplicated in space-oriented indexing, because the partition boundaries are not fixed and are generated from the distribution of data. This indicates that data-oriented indexing structures require less memory than their counterparts. However, the irregular decomposition in R-tree makes parallelization more difficult than space-oriented indexing techniques. In this dissertation, we have developed both R-tree parallel bulk loading and data-parallel tree traversal on the GPU.

There are several directions for the future work on spatial indexing. First, it is very useful to study how to reduce memory footprint in space-oriented indexing techniques, e.g., Grid-file. A possible solution is to develop cost models to determine the optimal configuration parameters for

space-oriented indexing techniques (such as [91]). Another possible solution is to develop multi-level Grid-file techniques to extend the single-level Grid-file techniques, which can potentially reduce memory footprint by aggregating grid cells in lower level into higher level grid cells. Second, hybrid indexing technique that can combine both data-oriented and space-oriented using data-parallel design can also be a future work direction. The hybrid indexing technique can potentially take advantages of the two types of indexing techniques. Another future work direction is to develop supports for more types of query processing, such as *k*-Nearest-Neighbor query.

Partition strategies have been introduced and data-parallel designs and implementations have been developed for spatial indexing in distributed computing environments. However, the distributed indexing techniques developed in this work mainly focus on supporting efficient distributed spatial join. For the future work, we would like to investigate on developing distributed indexing techniques for additional Big Data platforms, such as Apache Spark and Apache HBase. We also would like to extend our current designs to address the challenge in order to support the practical needs of real world applications. Furthermore, our current design mainly focuses on managing spatial data that are either static or infrequently updated. As such, another future direction is to develop distributed indexing support that is capable of dealing with dynamic data. The dynamic indexing techniques can be used to manage live streaming data with spatial context, such as geo-tagged tweets.

6.2.2 Spatial Join Techniques

In this dissertation work, spatial join techniques have been developed to support large-scale spatial data processing. Both single-node parallel spatial join and multi-node distributed spatial join have been studied. The spatial join processing is first scaled up on a single node and then scaled out across multiple nodes, which have achieved significant performance improvement. In single-node spatial join, both spatial filter and refinement phases have been developed with data-parallel designs to take advantages of existing parallel hardware, i.e., multi-core CPU and GPU. For distributed spatial join, two frameworks, including partition based spatial join and broadcast based spatial join, have been introduced for symmetric and asymmetric datasets.

For the future work, first, we would like to investigate on how to further improve the efficiency of our spatial join techniques for large-scale spatial data processing on Big Data platforms, including SpatialSpark and LDE. We believe there is still space to improve spatial join processing in distributed environments. For example, incorporating selectivity estimation into the spatial join framework can help generating better workload as well as scheduling for distributed processing. Second, for practical applications, how to adapt general designs to specific application is also very important. Different platforms may have different constraints which can potentially break the assumption made by the design or even completely change the design. As part of our future work, we would like to leverage our experiences to provide insights and suggestions for designing large-scale spatial join processing for different platforms. Third, our experiments show that broadcast based spatial join is more efficient than spatial partition based spatial join in many cases. As such, another future direction can be developing a hybrid approach that is able to take advantages of broadcast based spatial join but requires less memory footprint.

Appendix A. Parallel Primitives

Parallel primitives refer to a collection of fundamental algorithms that can be run on parallel machines. The behaviors of popular parallel primitives on one dimensional (1D) arrays or vectors are well-understood. Parallel primitives usually are implemented on top of native parallel programming languages (such as CUDA) but provide a set of simple yet powerful interfaces (or APIs) to end users. Technical details are hidden from end users and many parameters that are required by native programming languages are fine-tuned for typical applications in parallel libraries so that users do not need to specify such parameters explicitly.

On the other hand, such APIs usually use template or generic based programming techniques in a way similar to the well known C++ Standard Template Library (STL) so that the same set of APIs can be used for many data types. Due to the nature of high-level abstractions, the APIs may not be the most efficient ones when compared with handwritten programs using native programming languages with fine-tuned parameters. However, the APIs usually provide good tradeoffs between coding complexity and code efficiency. For example, most of the parallel primitives provided by the Thrust library are very similar to their STL counterparts and are very appealing to experienced STL users. The high level abstractions also bring significant portability. This unique feature further makes parallel primitives based algorithm developments attractive when compared with using native programming languages (e.g., CUDA) directly. In the rest of this appendix, we will introduce several commonly used parallel primitives.

(1) *scan*. The s*can* primitive computes the cumulative sum of an array. Both the inclusive and exclusive scans are possible. For example, *exclusive_scan*([3,2,0,1]) = [0,3,5,5] while

inclusive_scan([3,2,0,1]) = [3,5,5,6]. The Scan primitive can also take a user defined associative binary function to replace the default plus/sum binary function.

(2) *copy*, *copy_if*, *remove* and *remove_if*. *copy* moves groups of elements from one location to another location, typically in two different arrays. The *copy_if* primitive takes an additional unary function as a parameter to tell whether the corresponding array element should be copied to the output array or not. Similarly *remove* and *remove_if* remove groups of elements within an array with or without an optional binary predict function. *remove* and *remove_if* are typically applied in-place which means that the input arrays can be the same as output arrays to save memory. Note that compacted arrays after applying *remove* and *remove_if* primitives can be resized to reduce memory footprints.

(3) *transform*. The basic form of *transform* applies a unary function to each element of an input array and stores the result in the corresponding position in an output array. *transform* is more general than *copy* as it allows a user defined operation to be applied to array elements rather than simply copying. In many other systems, the *transform* primitive is also called *map*, such as *map* in MapReduce and *map/flatMap* in Spark.

(4) *scatter*. *scatter* copies elements from a source range of an input array into an output array according to a map. For example, *scatter*([3,0,2],[12,4,8],[*,*,*,*,*,*]) = ([4,*,8,*,12,*]). Note * values are those unchanged in the third array. Clearly when there is a one-to-one map between the inputs and outputs such as the Z-order transformation in our application, the output array will have no * values.

(5) *reduce*. *reduce* is an aggregation operator that produces reduction based on a binary function. For example, *reduce*([1, 2, 2, 1], +) = 6, where in this example, the plus operator is applied and the final results is the sum of all four numbers. *reduce_by_key* is an improvement over the original *reduce* operation. Instead of generating a single reduction result, only values that have the same key will be reduced. For example, *reduce_by_key*([1, 1, 1, 2], [1, 2, 2, 1], +) = [(1,5), (2, 1)]. The array of [1, 1, 1, 2] contains reduction keys and only those values have the same keys will be added together.

Appendix B. Publication during PhD Study

1. [Refereed Workshop] Simin You, Jianting Zhang and Le Gruenwald (2015). Spatial Join Query Processing in Cloud: Analyzing Design Choices and Performance Comparisons. *To appear in High Performance Computing for Big Data Workshop (HPC4BD) 2015.*
2. [Refereed Conference] Jianting Zhang, Simin You and Le Gruenwald. A Lightweight Distributed Execution Engine for Large-Scale Spatial Join Query Processing. *IEEE International Congress on Big Data 2015*.
3. [Refereed Workshop] Jianting Zhang, Simin You and Le Gruenwald (2015). Tiny GPU Cluster for Big Spatial Data: A Preliminary Performance Evaluation. *International Workshop on High-Performance Big Data Computing (HPBDC) 2015*.
4. [Invited Journal (non-refereed)] Jianting Zhang, Simin You and Le Gruenwald. Large-Scale Spatial Data Processing on GPUs and GPU-Accelerated Clusters. *ACM SIGSPATIAL Special, 6(3), pp. 27-34*.
5. [Refereed Workshop] Simin You, Jianting Zhang and Le Gruenwald. Large-Scale Spatial Join Query Processing in Cloud. *IEEE ICDE CloudDM International Workshop 2015*.
6. [Refereed Workshop] Simin You, Jianting Zhang and Le Gruenwald. Scalable and Efficient Spatial Data Management on Multi-Core CPU and GPU Clusters: A Preliminary Implementation based on Impala. *IEEE ICDE HardBD International Workshop 2015*.
7. [Refereed Journal] Jianting Zhang, Simin You and Le Gruenwald. Parallel Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs. *Information Systems (Elsevier journal), 2014*.
8. [Refereed Workshop] Jianting Zhang, Simin You and Le Gruenwald. Data Parallel Quadtree Indexing and Spatial Query Processing of Complex Polygon Data on GPUs. *VLDB ADMS International Workshop 2014*.
9. [Refereed Conference] Jianting Zhang, Simin You and Le Gruenwald. High-Performance Spatial Query Processing on Big Taxi Trip Data using GPGPUs. *IEEE International Congress on Big Data 2014*.
10. [Refereed Journal] Jianting Zhang and Simin You. High-Performance Quadtree Constructions on Large-Scale Geospatial Rasters Using GPGPU Parallel Primitives. *International Journal of Geographical Information Sciences (IJGIS) 2013*.
11. [Refereed Workshop] Simin You, Jianting Zhang, and Le Gruenwald. GPU-based Spatial Indexing and Query Processing Using R-Trees. *ACM SIGSPATIAL BigSpatial International Workshop2013*.
12. [Refereed Conference] Jianting Zhang and Simin You. Constructing Natural Neighbor Interpolation Based Grid DEM Using CUDA. *COM.Geo Conference 2012*.

13. [Refereed Workshop] Jianting Zhang, Simin You and Le Gruenwald. High-Performance Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs. *ACM CIKM DOLAP International Workshop 2012*.
14. [Refereed Workshop] Jianting Zhang, Simin You and Le Gruenwald. U2STRA: High-Performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs. *ACM CDMW International Workshop 2012*.
15. [Refereed Workshop] Jianting Zhang and Simin You. CudaGIS: Report on the Design and Realization of a Massive Data Parallel GIS on GPUs. *ACM SIGSPATIAL IWGS International Workshop 2012*.
16. [Refereed Workshop] Jianting Zhang and Simin You. Speeding up Large-Scale Point-in-Polygon Test Based Spatial Join on GPUs. *ACM SIGSPATIAL BigSpatial International Workshop 2012*.
17. [Refereed Conference] Jianting Zhang, Simin You and Le Gruenwald. Parallel Quadtree Coding of Large-Scale Raster Geospatial Data on GPGPUs. *ACM SIGSPATIAL GIS 2011*.
18. [Refereed Workshop] Simin You and Jianting Zhang. Efficient Histogramming of Large-Scale Geospatial Rasters in Support of Web-Based Queries. *ACM SIGSPATIAL HPDGIS International Workshop 2011*.
19. [Refereed Conference/Book Chapter] Jianting Zhang and Simin You. Supporting Web-based Visual Exploration of Large-Scale Raster Geospatial Data Using Binned Min-Max Quadtree. *SSDBM Conference 2010*.
20. [Refereed Conference] Jianting Zhang, Simin You and Le Gruenwald. Indexing Large-Scale Raster Geospatial Data Using Massively Parallel GPGPU Computing. *ACM SIGSPATIAL GIS Conference 2010*.
21. [Refereed Conference] Jianting Zhang, Simin You, Li Chen, and Cynthia Chen. A Hybrid Approach to Segment-Type Coding of New York City Traffic Data. *COM.Geo Conference 2010*.
22. [Refereed Conference] Jianting Zhang and Simin You. Dynamic Tiled Map Services: Supporting Query-Based Visualization of Large-Scale Raster Geospatial Data. *COM.Geo Conference 2010*.

Reference

[1] A Benchmark for Multidimensional Index Structures: *http://www.mathematik.uni-marburg.de/~rstar/benchmark/*.

[2] Aboulnaga, A. and Aref, W. 2001. Window query processing in linear quadtrees. *Distributed and Parallel Databases*. (2001).

[3] Aji, A. et al. 2013. Demonstration of Hadoop-GIS. *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL'13* (New York, New York, USA, 2013), 518–521.

[4] Aji, A. et al. 2013. Hadoop GIS: a high performance spatial data warehousing system over mapreduce. *Proc. VLDB Endow.* (2013).

[5] Alborzi, H. and Samet, H. 2007. Execution time analysis of a top-down R-tree construction algorithm. *Information Processing Letters*. 101, 1 (Jan. 2007), 6–12.

[6] Appuswamy, R. et al. 2013. Scale-up vs scale-out for Hadoop. *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC '13* (New York, New York, USA, 2013), 1–13.

[7] Audet, S. et al. 2013. Robust and efficient polygon overlay on parallel stream processors. *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL'13*. (2013), 294–303.

[8] Bakkum, P. and Skadron, K. 2010. Accelerating SQL database operations on a GPU with CUDA. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units - GPGPU '10* (New York, New York, USA, 2010), 94.

[9] Bandi, N. et al. 2007. Fast computation of spatial selections and joins using graphics hardware. *Information Systems*. 32, 8 (2007), 1073–1100.

[10] Bandi, N. et al. 2004. Hardware acceleration in commercial databases: A case study of spatial operations. *Proceedings of the Thirtieth international conference on Very large data bases* (2004), 1021–1032.

[11] Batista, V.H.F. et al. 2010. Parallel geometric algorithms for multi-core computers. *Computational Geometry: Theory and Applications*. 43, 8 (2010), 663–677.

[12] Beckmann, N. et al. 1990. The R*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD Record*.

[13] Bentley, J.L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM*. 18, 9 (Sep. 1975), 509–517.

[14] Bittorf, M. and Bobrovytsky, T. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. *CIDR* (2015).

[15] Brinkhoff, T. et al. 1993. Efficient processing of spatial joins using R-trees. *Proceedings of the 1993 ACM SIGMOD international conference on Management of data - SIGMOD '93* (New York, New York, USA, 1993), 237–246.

[16] Brinkhoff, T. et al. 1996. Parallel processing of spatial joins using R-trees. *Data Engineering, 1996. Proceedings of the Twelfth International Conference on* (1996), 258–265.

[17] Chung, K.L. et al. 2007. Efficient algorithms for coding Hilbert curve of arbitrary-sized image and application to window query. *Information Sciences*. 177, 10 (2007), 2130–2151.

[18] Comer, D. 1979. Ubiquitous B-Tree. *ACM Computing Surveys*. 11, 2 (Jun. 1979), 121–137.

[19] Dean, J. and Ghemawat, S. 2004. MapReduce - Simplified Data Processing on Large Clusters. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (2004), 137–149.

[20] Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified Data Processing on Large Clusters. *OSDI '04* (2004), 137–150.

[21] Dieker, S. and Guting, R.H. 2000. Plug and play with query algebras: SECONDO, a generic DBMS development environment. *Proceedings 2000 International Database Engineering and Applications Symposium (Cat. No.PR00789)*. (2000).

[22] Dittrich, J.-P. and Seeger, B. 2000. Data redundancy and duplicate detection in spatial join processing. *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*. Icde (2000), 535–546.

[23] Eldawy, A. et al. 2013. CG_Hadoop. *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL'13* (New York, New York, USA, 2013), 284–293.

[24] Eldawy, A. et al. 2015. Spatial Hadoop : A MapReduce Framework Supporting Spatial Data. *IEEE International Conference on Data Engineering, ICDE 2015* (2015).

[25] Eldawy, A. and Mokbel, M. 2013. A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data. *Proceedings of the VLDB Endowment*. 6, 12 (2013).

[26] Eldawy, A. and Mokbel, M.F. 2014. Pigeon: A spatial MapReduce language. *2014 IEEE 30th International Conference on Data Engineering*. (Mar. 2014), 1242–1245.

[27] Fang, R. et al. 2007. GPUQP: Query Co-Processing Using Graphics Processors. *Proceedings of the International Conference on Management of Data (SIGMOD)*. (2007), 1061–1063.

[28] Finkel, R.A. and Bentley, J.L. 1974. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*. 4, 1 (1974), 1–9.

[29] Fox, A. et al. 2013. Spatio-temporal indexing in non-relational distributed databases. *2013 IEEE International Conference on Big Data* (Oct. 2013), 291–299.

[30] Gaede, V. and Günther, O. 1998. Multidimensional access methods. *ACM Computing Surveys*. 30, 2 (Jun. 1998), 170–231.

[31] Gargantini, I. 1982. An effective way to represent quadtrees. *Communications of the ACM*. 25, 12 (Dec. 1982), 905–910.

[32] Govindaraju, N.K. et al. 2004. Fast computation of database operations using graphics processors. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04* (2004), 215.

[33] Gowanlock, M. and Casanova, H. 2014. Distance threshold similarity searches on spatiotemporal trajectories using GPGPU. *2014 21st International Conference on High Performance Computing (HiPC)* (Dec. 2014), 1–10.

[34] Gowanlock, M. and Casanova, H. 2015. Indexing of Spatiotemporal Trajectories for Efficient Distance Threshold Similarity Searches on the GPU. *IEEE IPDPS* (2015).

[35] Guttman, A. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84* (New York, New York, USA, Jun. 1984), 47.

[36] He, B. et al. 2008. Relational joins on graphics processors. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08* (New York, New York, USA, 2008), 511.

[37] He, B. et al. 2009. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems*. 34, 4 (Dec. 2009), 1–39.

[38] Hennessy, J.L. and Patterson, D.A. 2006. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc.

[39] Hoel, E. and Samet, H. 1994. Data-Parallel Spatial Join Algorithms. *1994 International Conference on Parallel Processing (ICPP'94)* (Aug. 1994), 227–234.

[40] Hormann, K. and Agathos, A. 2001. The point in polygon problem for arbitrary polygons. *Computational Geometry*. 20, 3 (Nov. 2001), 131–144.

[41] Hu, Y. et al. 2012. Topological relationship query processing for complex regions in Oracle Spatial. *Proceedings of the 20th International Conference on Advances in Geographic Information Systems - SIGSPATIAL '12*. 1 (2012), 3.

[42] Huang, Y. et al. 1997. Spatial joins using R-trees: Breadth-first traversal with global optimizations. *VLDB '97 Proceedings of the 23rd International Conference on Very Large Data Bases* (1997), 396–405.

[43] Jacox, E.H. and Samet, H. 2003. Iterative spatial join. *ACM Transactions on Database Systems*.

[44] Jacox, E.H. and Samet, H. 2007. Spatial join techniques. *ACM Transactions on Database Systems*. 32, 1 (Mar. 2007), 7–es.

[45] Kalojanov, J. and Slusallek, P. 2009. A parallel algorithm for construction of uniform grids. *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09* (New York, New York, USA, 2009), 23.

[46] Kamel, I. and Faloutsos, C. 1993. On packing R-trees. *Proceedings of the second international conference on Information and knowledge management - CIKM '93* (New York, New York, USA, Dec. 1993), 490–499.

[47] Kamel, I. and Faloutsos, C. 1992. Parallel R-trees. *Proceedings of the 1992 ACM SIGMOD international conference on Management of data - SIGMOD '92* (New York, New York, USA, Jun. 1992), 195–204.

[48] Khlopotine, A.B. et al. 2013. A Variant of Parallel Plane Sweep Algorithm for Multicore Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 32, 6 (Jun. 2013), 966–970.

[49] Kim, J. et al. 2012. A Performance Study of Traversing Spatial Indexing Structures in Parallel on GPU. *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems* (Jun. 2012), 855–860.

[50] Kim, J. et al. 2013. Parallel multi-dimensional range query processing with R-trees on GPU. *Journal of Parallel and Distributed Computing*. 73, 8 (Apr. 2013), 1195–1207.

[51] Koudas, N. et al. 1996. Declustering spatial databases on a multi-computer architecture. *Advances in Database Technology — EDBT '96*. 8958546, (1996).

[52] Kumar, K.A. et al. 2014. Optimization Techniques for " Scaling Down " Hadoop on Multi-Core , Shared-Memory Systems. *EDBT* (2014), 13–24.

[53] Lai, S. et al. 2000. A Design of Parallel R-tree on Cluster of Workstations. *Databases in Networked Information Systems*. (2000), 119–133.

[54] Lemire, D. and Boytsov, L. 2013. Decoding billions of integers per second through vectorization. *Software - Practice and Experience*.

[55] Leutenegger, S.T. et al. 1997. STR: a simple and efficient algorithm for R-tree packing. *Proceedings 13th International Conference on Data Engineering* (1997), 497–506.

[56] Li, J. et al. 2007. Point-in-polygon tests by convex decomposition. *Computers & Graphics*. 31, 4 (Aug. 2007), 636–648.

[57] Li, S. et al. 2015. Pyro: A Spatial-Temporal Big-Data Storage System. *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), 97–109.

[58] Lieberman, M.D. et al. 2008. A Fast Similarity Join Algorithm Using Graphics Processing Units. *2008 IEEE 24th International Conference on Data Engineering* (Apr. 2008), 1111–1120.

[59] Liu, E.S. and Theodoropoulos, G.K. 2009. An Approach for Parallel Interest Matching in Distributed Virtual Environments. *2009 13th IEEEACM International Symposium on Distributed Simulation and Real Time Applications*. (2009), 57–65.

[60] Lo, M.-L. and Ravishankar, C. V 1996. Spatial hash-joins. *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96* (New York, New York, USA, 1996), 247–258.

[61] Lu, J. and Guting, R.H. 2012. Parallel Secondo: Boosting Database Engines with Hadoop. *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. (Dec. 2012), 738–743.

[62] Luo, L. et al. 2010. An effective GPU implementation of breadth-first search. *Proceedings of the 47th Design Automation Conference on - DAC '10* (New York, New York, USA, 2010), 52.

[63] Luo, L. et al. 2012. Parallel implementation of R-trees on the GPU. *17th Asia and South Pacific Design Automation Conference* (Jan. 2012), 353–358.

[64] McCool, M. et al. 2012. *Structured parallel programming patterns for efficient computation*. Morgan Kaufmann Publishers Inc.

[65] McKenney, M. et al. 2011. Geospatial overlay computation on the GPU. *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '11*. (2011), 473.

[66] McKenney, M. and McGuire, T. 2009. A parallel plane sweep algorithm for multi-core systems. *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems GIS 09*. (2009), 392.

[67] Mokbel, M.F. et al. 2003. Analysis of multi-dimensional space-filling curves. *GeoInformatica*. 7, 3 (2003), 179–209.

[68] Mutenda, L. and Kitsuregawa, M. 1999. Parallel R-tree spatial join for a shared-nothing architecture. *Proceedings 1999 International Symposium on Database Applications in Non-Traditional Environments (DANTE'99) (Cat. No.PR00496)* (1999), 423–430.

[69] Naami, K.M. Al et al. 2014. GISQF: An Efficient Spatial Query Processing System. *2014 IEEE 7th International Conference on Cloud Computing* (Jun. 2014), 681–688.

[70] Nievergelt, J. et al. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*. 9, 1 (Jan. 1984), 38–71.

[71] Nishimura, S. et al. 2011. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. *2011 IEEE 12th International Conference on Mobile Data Management* (Jun. 2011), 7–16.

[72] Nobari, S. et al. 2013. TOUCH: in-memory spatial join by hierarchical data-oriented partitioning. *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. (2013).

[73] Olma, M. et al. 2013. BLOCK: Efficient Execution of Spatial Range Queries in Main-Memory. *EPFL-REPORT-190731* (2013).

[74] Papadopoulos, A. and Manolopoulos, Y. 2003. Parallel bulk-loading of spatial data. *Parallel Computing*. 29, 10 (Oct. 2003), 1419–1444.

[75] Patel, J.M. and DeWitt, D.J. 2000. Clone join and shadow join. *Proceedings of the eighth ACM international symposium on Advances in geographic information systems - GIS '00* (New York, New York, USA, 2000), 54–61.

[76] Patel, J.M. and DeWitt, D.J. 1996. Partition based spatial-merge join. *ACM SIGMOD Record*. 25, 2 (Jun. 1996), 259–270.

[77] Polychroniou, O. and Ross, K. a. 2014. Vectorized Bloom filters for advanced SIMD processors. *Proceedings of the Tenth International Workshop on Data Management on New Hardware - DaMoN '14* (New York, New York, USA, 2014), 1–6.

[78] Prasad, S.K. et al. 2015. A vision for GPU-accelerated parallel computation on geo-spatial datasets. *SIGSPATIAL Special*. 6, 3 (Apr. 2015), 19–26.

[79] Rauhe, H. et al. 2013. Multi-level Parallel Query Execution Framework for CPU and GPU. *17th East European Conference, ADBIS 2013, Genoa, Italy, September 1-4, 2013* (2013), 330–343.

[80] Ray, S. et al. 2013. A parallel spatial data analysis infrastructure for the cloud. *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL'13*. (2013), 274–283.

[81] Ray, S. et al. 2014. Skew-resistant parallel in-memory spatial join. *Proceedings of the 26th International Conference on Scientific and Statistical Database Management - SSDBM '14* (New York, New York, USA, 2014), 1–12.

[82] Samet, H. 2006. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc.

[83] Samet, H. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*.

[84] Schnitzer, B. and Leutenegger, S.T. 1998. Master-client R-trees: a new parallel R-tree architecture. *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management* (Jul. 1998), 68–77.

[85] Sellis, T.K. et al. 1987. The R+-tree: A Dynamic Index for Multi-dimensional Objects. *International Conference on Very Large Databases (VLDB)* (1987), 507–518.

[86] Shamos, M.I. and Hoey, D. 1976. Geometric intersection problems. *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)* (Oct. 1976), 208–215.

[87] Šidlauskas, D. et al. 2009. Trees or grids? *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '09* (New York, New York, USA, 2009), 236.

[88] Šidlauskas, D. and Jensen, C.S. 2014. Spatial joins in main memory: implementation matters! *Proceedings of the VLDB Endowment*. 8, 1 (Sep. 2014), 97–100.

[89] Silvestri, C. et al. Computing Iterated Spatial Joins on GPUs. *Technical Report http://madalgo.au.dk/fileadmin/madalgo/OA_PDF_s/C337.pdf*. X.

[90] Simion, B. et al. 2012. Speeding up Spatial Database Query Execution using GPUs. *Procedia Computer Science*. 9, Wepa (Jan. 2012), 1870–1879.

[91] Tauheed, F. et al. 2015. Configuring Spatial Grids for Efficient Main Memory Joins. *30th British International Conference on Databases, BICOD 2015* (2015), 199–205.

[92] Tauheed, F. et al. 2015. THERMAL-JOIN. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15* (New York, New York, USA, 2015), 939–950.

[93] Tu, Y. et al. 2013. Data management systems on GPUs. *Proceedings of the 25th International Conference on Scientific and Statistical Database Management - SSDBM* (New York, New York, USA, 2013), 1.

[94] Van, L.H. and Takasu, A. 2015. An Efficient Distributed Index for Geospatial Databases. *Dexa* (Berlin, Heidelberg, 2015), 28–42.

[95] Venetis, P. and Gonzalez, H. 2011. Hyper-local, directions-based ranking of places. *Proceedings of the VLDB Endowment*. 4, 5 (2011), 290–301.

[96] Vo, H. et al. 2014. SATO: A Spatial Data Partitioning Framework for Scalable Query Processing. *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL '14* (2014).

[97] Wang, B. et al. Parallel R-tree search algorithm on DSVM. *Proceedings. 6th International Conference on Advanced Systems for Advanced Applications* 237–244.

[98] Wang, F. et al. 2011. Hadoop-GIS: A High Performance Spatial Query System for Analytical Medical Imaging with MapReduce. *Technical report, Emory University*. (2011).

[99] Wang, K. et al. 2012. Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems. *Proc. VLDB Endow.* 5, 11 (Jul. 2012), 1543–1554.

[100] Ward, P.G.D. et al. 2014. Real-time continuous intersection joins over large sets of moving objects using graphic processing units. *The VLDB Journal*. 23, 6 (Dec. 2014), 965–985.

[101] Yampaka, T. and Chongstitvatana, P. 2012. Spatial join with r-tree on graphics processing units. *8th International Conference on Computing and Information Technology* (2012).

[102] Yang, K. et al. 2007. In-memory grid files on graphics processors. *Proceedings of the 3rd international workshop on Data management on new hardware - DaMoN '07*. (2007), 1.

[103] Yi, K. 2008. *Encyclopedia of Algorithms*. Springer US.

[104] Yu, B. et al. 2011. Parallel Range Query Processing on R-Tree with Graphics Processing Unit. *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing* (Dec. 2011), 1235–1242.

[105] Yuan, Y. et al. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment*. 6, 10 (Aug. 2013), 817–828.

[106] Zaharia, M. et al. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. (2012), 2–2.

[107] Zhang, S. et al. 2009. Sjmr: Parallelizing spatial join with mapreduce on clusters. *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (2009), 1–8.

[108] Zhong, Y. et al. 2012. Towards Parallel Spatial Query Processing for Big Spatial Data. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (May 2012), 2085–2094.

[109] Zhou, J. and Ross, K.A. 2002. Implementing Database Operations Using SIMD Instructions. *International Conference on Management of Data*. (2002), 145.

[110] Zhou, T. et al. 2013. Point-polygon topological relationship query using hierarchical indices. *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL'13*. (2013), 562–565.

[111] Zhou, X. et al. 1998. Data partitioning for parallel spatial join processing. *GeoInformatica*. 204, (1998), 175–204.