Large-Scale Spatial Data Management on Modern Parallel and Distributed Platforms

Thesis Proposal

Abstract

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. We propose to develop 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 cluster computers, to achieve both efficiency and scalability. After introducing background on spatial data management and modern parallel and distributed systems, we present our preliminary data-parallel designs for spatial indexing and spatial join queries on both multi-core CPUs and GPUs for high efficiency as well as their integrations with Big Data systems for better scalability. Following the discussions on some experiment results on real datasets to demonstrate the effectiveness and efficiency of the proposed designs, the roadmap of dissertation work is subsequently outlined.

Simin You

1/26/2015
# Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction .................................................................</td>
</tr>
<tr>
<td>2</td>
<td>Background and Related Work ...............................................</td>
</tr>
<tr>
<td>2.1</td>
<td>Modern Parallel and Distributed Platforms ................................</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Single-Node Platforms .....................................................</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Multi-Node Distributed Platforms .........................................</td>
</tr>
<tr>
<td>2.2</td>
<td>Spatial Index ........................................................................</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Grid-Files ............................................................................</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Quadtrees ............................................................................</td>
</tr>
<tr>
<td>2.2.3</td>
<td>R-trees ................................................................................</td>
</tr>
<tr>
<td>2.3</td>
<td>Spatial Join Query ...............................................................</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Problem Definition ................................................................</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Plane-Sweep based Spatial Join ...............................................</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Indexed Nested-loop Spatial Join .............................................</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Synchronized Index Traversal based Spatial Join .......................</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Partition Based Spatial Join ..................................................</td>
</tr>
<tr>
<td>3</td>
<td>Proposed Designs and Implementations ......................................</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview ...............................................................................</td>
</tr>
<tr>
<td>3.2</td>
<td>Parallel Spatial Indexing .......................................................</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Spatial Data Layout ................................................................</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Parallel Grid-File based Indexing for MBRs ...........................</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Parallel R-tree based Indexing ...............................................</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Parallel Quadtree based Indexing for Point Data ....................</td>
</tr>
<tr>
<td>3.3</td>
<td>Parallel Spatial Join Design ...................................................</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Single-Node Parallel Spatial Join ...........................................</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Multi-Node Parallel Spatial Join ............................................</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Proposed Prototype Systems ....................................................</td>
</tr>
<tr>
<td>3.4</td>
<td>Preliminary Results ..................................................................</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Preliminary Results of Parallel Spatial Data Management on Single-Node</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Preliminary Results on Multi-Node Parallel Spatial Join .............</td>
</tr>
<tr>
<td>4</td>
<td>Summary and Research Plan .....................................................</td>
</tr>
<tr>
<td>5</td>
<td>Reference ..............................................................................</td>
</tr>
</tbody>
</table>
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 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 developing 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 technologies, e.g., MapReduce [9] 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 techniques for managing spatial data 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, 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 propose to develop techniques that can efficiently manage large-scale spatial data on modern parallel and distributed platforms. First of all, we will develop new parallel designs, including parallel spatial indexing and query processing techniques, for large-scale spatial data management. Second, we will investigate on how to implement such parallel designs using parallel primitives that are efficiently supported by many modern parallel

\(^1\) http://data.gbif.org

\(^2\) http://hadoop.apache.org
platforms to achieve interoperability and productivity. Last but not least, we will develop relevant techniques to scale out spatial data processing to cluster computers that are increasingly available in Cloud Computing.

2 Background and Related Work

In this section, we introduce the background and related work of the proposed research. First, we will introduce existing modern parallel and distributed platforms that are related to the proposed research. Second, three important spatial indexing techniques related to our proposed research are briefly discussed. Finally, we will define spatial join query in the context of the proposed research. Related works on parallel and distributed spatial indexing and spatial join are briefly discussed in the respective subsections.

2.1 Modern Parallel and Distributed Platforms

The recent development of parallel computing 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 section.

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.

Figure 1 Parallel and Distributed Platforms
2.1.1.1 Multi-core CPUs

While clock frequency on a single CPU core is nearly reaching the 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. 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 nowadays. However, there is still a significant gap between hardware and software as many software packages have not fully taken advantage of parallel hardware. To alleviate this gap, various parallel programming models have been developed. A common approach to utilize multi-core systems is using the thread model, such as those based on OpenMP\textsuperscript{3} and Intel Threading Building Blocks (TBB\textsuperscript{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. We call this level of parallelism as \textit{task level parallelism}, where computation is divided into tasks and executed independently among threads.

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. 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 \(p*\). 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. However, using SIMD computing power for spatial data processing is challenging for two reasons. First, SIMD instructions are usually restricted, and we need to 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 serial execution. Thus, memory access pattern in spatial data processing needs to be considered in order to achieve good performance.

Figure 2 shows an 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

\textsuperscript{3} http://openmp.org

\textsuperscript{4} http://threadingbuildingblocks.org
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 VPU s 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 computing power.

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 proposal, 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

\(^5\) https://developer.nvidia.com/what-cuda
parallel portions of an application execution 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 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 communication cost between CPU main memory and GPU memory is expensive. GPUs are attached via PCI-E buses and data must be first transferred from CPU memory to GPU memory before performing computation. 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 summing 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.
2.1.2 Multi-Node Distributed 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 the general public. New distributed computing tools, such as MapReduce and its open source implementation Hadoop, have made it much easier to develop and deploy parallel tasks on cluster computers provided by Cloud Computing vendors (such as Amazon EC2\(^7\)). We next review two categories of distributed Big Data platforms, one is based on disk and the other 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 based on MapReduce/Hadoop

MapReduce [9] is a parallel 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

---

\(^6\) http://www.mpi-forum.org

\(^7\) http://aws.amazon.com/ec2/
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.

\[
\text{map: } (key_1, value_1) \rightarrow \text{list}(key_2, value_2) \\
\text{reduce: } (key_2, \text{list}(value_2)) \rightarrow \text{list}(value_3)
\]

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. 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 sacrifices expressive power and performance. Another issue of Hadoop based system 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 [3]. 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 [47] and Cloudera Impala [5], 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) [47] 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\(^8\) for control-intensive communication and Netty\(^9\) 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 distributed disk storage which also helps to achieve fault tolerance. Although hardware failures are rare in small clusters [22], 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\(^10\).

When comparing Spark with Hadoop, 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 considered as a replacement of Apache Hive\(^11\) (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

\(^8\)http://akka.io/

\(^9\)http://netty.io/

\(^10\)https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html

\(^11\)https://hive.apache.org
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 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 Index

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 [37]. In this section, we briefly introduce three major spatial indexes that are related to our proposed research, i.e., Grid-files, Quadtrees and R-Trees. The major characteristics of the three categories of spatial indexes are tabulated in Table 1. 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

<table>
<thead>
<tr>
<th></th>
<th>Grid file</th>
<th>Quadtree</th>
<th>R-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition Strategy</td>
<td>space-oriented</td>
<td>space-oriented</td>
<td>data-oriented</td>
</tr>
<tr>
<td>Hierarchical Structure</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Parallelization friendly</td>
<td>Good</td>
<td>Medium</td>
<td>Poor</td>
</tr>
<tr>
<td>Skewness Handling</td>
<td>Poor</td>
<td>Medium</td>
<td>Good</td>
</tr>
</tbody>
</table>

2.2.1 Grid-Files

Grid-file [30] 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 an 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 needs to be stored in order to locate each grid cell correctly. On the contrary, 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 proposed research, we prefer uniform grid file for simplicity. We will use grid-file refer to uniform grid-file hereafter.
Unlike tree based hierarchical structures, a grid-file uses a flat structure that splits the space into grid cells, where each grid cell is a subspace. Also, the simplicity of grid file has demonstrated its efficiency on modern hardware, comparing with tree based indexes [40]. Previous works [19, 45] have shown that both index construction and query processing can be significantly improved by using grid-file indexing on GPUs. Both [19] and [45] optimized the ray-tracing application using grid-file index on GPUs. Unlike previous works that focus on visualization, we plan to exploit the potentials of utilizing parallel grid-file indexing for spatial data management. We also plan to develop data-parallel designs using parallel primitives for grid-file based indexing, especially for supporting spatial join processing (Section 2.3).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{grid bribery.png}
\caption{Spatial Index Examples}
\end{figure}

### 2.2.2 Quadtrees

Quadtree [12] is a tree structure that is used for indexing spatial objects in 2-D space. It behaves similar to binary trees in 1-D space. While there are many Quadtree variants, in this proposed research, we use the term Quadtree to refer to Region Quadtree [37]. 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 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 [37] which is a mapping based on Z-order [37]. The mapping can be realized by using 0, 1, 2, 3 to represent NW, NE, SW, SE nodes respectively [13]. For example, the left most node in the last level of Figure 5b is (enclosed in the dotter square) as 0030. The regular splitting pattern of Quadtree is suitable for data-parallel designs. For example, the work in [15] 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. Thus, it is technically challenging to develop a parallel Quadtree structure that can fully exploit parallelism. In this proposal, we will design 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-trees [14, 46] are well known spatial indexing techniques and have been widely adopted in many applications for indexing 2-D or higher dimensional spatial data. Similar to B-tree [8], R-trees are also balanced search trees but are adapted for multi-dimensional data. The key idea of R-trees is to group nearby objects and represent their aggregated spatial extent as an MBR. Unlike Quadtrees that generate 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 dataset, bulk loading might be more efficient. In bulk loading, R-tree is constructed from the whole dataset typically by sorting and hierarchically aggregating MBRs bottom-up [33]. 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 parallel DFS based batched 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 [26] suggested BFS based query processing, which can be more efficient on parallel hardware especially GPUs.

In this proposal, we would like to improve parallel R-tree construction using parallel primitives (Section 3.2.3). The design is portable across multiple parallel platforms and improves the works in [26, 42]. We also propose to develop parallel primitive based designs for query processing which can serve as a module for efficient spatial join query processing.

2.3 Spatial Join Query
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” 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).
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

\[
R \Join_{\text{cond}} S = \{(r, s) | r \in R, s \in S, \text{relation}(r, s) \text{ is held}\},
\]

where \text{relation} is a spatial relationship (usually a spatial predicate) between two spatial objects. Figure 6 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 7, the naïve approach requires twelve intersection tests. However, if the space is indexed as partitions in advance and 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, \textit{filter-and-refinement} strategy is adopted in most of existing spatial join techniques [18].

The filter-and-refinement strategy divides spatial join processing into two steps, i.e., \textit{filter} and \textit{refinement}. In the filter step, spatial objects are first approximated by axis aligned MBRs, and then stored in the form of \( \langle \text{OID}, \text{MBR} \rangle \). Here \text{OID} is a pointer to the original spatial object and \text{MBR} refers to the extent of the spatial object. The approximated MBR representation saves expensive geometric computation on 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 pair process. Denoting \( OID_r \) and \( OID_s \) as pointers to original spatial objects in \( R \) and \( S \), the output of the filter step can be represented as a list of \( (OID_r, OID_s) \).

For the filter step, 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 has been given in Figure 7. Many other spatial relationship operators can be transformed into a 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 8.

The filter step prunes pairs that do not satisfy a spatial relationship but allows false positives because MBRs are used to approximate complex spatial objects. The refinement step completely removes all false positives from the previous step by testing the spatial relationship between two spatial objects based on their exact geometry. During the refinement step, original 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 step significantly impacts the overall performance of a spatial join. As such, most existing research has focused on optimizing the filter step in order to minimize false positives.

For the rest of Section 2.3, we will discuss four leading spatial join techniques, including planesweep, 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 our proposed research on parallel spatial joins in Section 3.3 while refer to [18] 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.
<table>
<thead>
<tr>
<th>Pre-processing</th>
<th>Plane-Sweep based</th>
<th>Indexed Nested-loop based</th>
<th>Synchronized Index Traversal based</th>
<th>Partition based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need Spatial Index?</td>
<td>Sort</td>
<td>Index construction</td>
<td>Index construction</td>
<td>Sort/Shuffle</td>
</tr>
<tr>
<td>Generate Duplicates in Output</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Support Data-Parallel Design?</td>
<td>Very Difficult (Sequential in nature)</td>
<td>Easy</td>
<td>Difficult (Due to irregular data access on trees)</td>
<td>Moderate</td>
</tr>
</tbody>
</table>

2.3.2 Plane-Sweep based Spatial Join

The classic plane-sweep algorithm [39] reports all intersections from two sets of rectangles (MBRs in spatial joins) efficiently and has been widely used in spatial databases and 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 [18], 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 [28]. 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 [28]. 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 [28], 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 [48]). 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. 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 batched 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 $\sigma$ 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 9 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.

```
Indexed_Nested_Loop_Join (R,S)
begin
    index_R ← Create_Index(R)
    foreach s ∈ S do
        results ← Index_Search(index_R, s)
        Report ((r,s)|r ∈ results )
    end
end
```

Figure 9 Indexed Nested-Loop Join Algorithm

Luo et al [26] implemented R-tree based batched 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. The other approach of parallelizing indexed nested loop join is to use spatial indices designed for parallelizing range queries. Kamel and Faloutsos [20] 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 [21] 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 [20], their design de-clustered R-tree nodes to multiple computer nodes. Another parallel R-tree structure on shared-nothing system is called Master-client R-tree was proposed by Schnitzer and Leutenegger [38], where 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 master node sequentially and then distributed it to client nodes to continue search in parallel. Lai et al [23] found that processing time on the master node in [38] 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 individually. As a result, the whole R-tree was distributed among all nodes and the bottleneck issue was solved. In Hadoop-GIS [2], the authors also adopted R-tree based nested loop spatial join. The technique first partitioned the input datasets by sampling [41], and then, shuffled the datasets according to the generated partitions. 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 [6] 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 [17] optimized the original R-tree join in [6] using BFS traversal that achieved better performance; however, it had a drawback on controlling the priority queue size during the traversal.

Brinkhoff et al [7] extended the sequential R-tree based join [6] to a shared-nothing parallel system. Similar to the sequential version, synchronized hierarchical traversal was used but sub-trees were sent to parallel processors for parallel processing. On shared-nothing parallel systems, in addition to CPU and I/O costs, communication cost is also a crucial factor. A challenge identified in [6] 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 [29]. 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 [11] implemented R-tree synchronized spatial join on Hadoop. When two datasets were indexed by R-tree, SpatialHadoop first generated the intersected partitions using global R-trees. For each partition pairs, synchronized spatial join was applied. For parallel R-tree joins on shared-memory systems, Yampaka and Chonstitivatana [44] described a GPU based spatial join using R-tree. They used the same design from [7] but distributed the MBR intersection tests on GPUs instead of
CPUs. During the spatial join, R-trees were bulk loaded before synchronized DFS traversal on the two R-trees until leaf nodes were reached.

Besides R-trees, Quadtrees have also been adopted in parallel spatial joins. Hoel and Samet [15] developed a data parallel spatial join using PMR-Quadtree [37] 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 [15] 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 (4) 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 [35]. Similar to other spatial join algorithms, PBSM included the filter and refinement steps. 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 10, partition 0 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 10, 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 \( N \times T \) tiles where \( N \times T \) was greater than \( P \). Subsequently the tiles were assigned to partitions in a Round-Robin manner (or using hashing). After the filter step, MBR pairs \((OID_r, OID_s)\) were generated for the refinement step. As duplicated MBRs were generated during partitioning, they could also be generated in the filter step and needed to be removed. This could be realized by sorting or Reference Point Method (RPM) technique suggested in [10]. 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 [34] 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 [34]. 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 pair for the refinement step in shadow joins.

**Figure 10 Tile-to-Partition and Skew Spatial Data**

SPINOJA [36] 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 [50] have implemented PBSM on a dedicated parallel machine. They improved the original PBSM partition function by using Z-order curve 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 [48] 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 [10]. Zhong et al [49] 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 in-memory spatial join technique based on Hilbert Space Filling Curve [37].

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 10). Lo and Ravishankar [25] suggested the Spatial Hash Join (SHJ) to address this issue. Instead of decomposing space into regular grid 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 outer dataset with the overlapping inner buckets. A recent technique called TOUCH [31] 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 retrieves 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. Partition based methods can also be adopted by distributed systems such as Hadoop. Both Hadoop-GIS [43] and SpatialHadoop [11] 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 support 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 achieved relatively easily, it requires more efforts to avoid/remove duplicates and achieve load balancing. For this reason, as indicated in the last row of Table 2, we rate the level of support for parallel designs as “Medium”.

3 Proposed Designs and Implementations

3.1 Overview
We propose to develop parallel designs of spatial data management at two levels. First, we would like to fully exploit the computing power on a single computing node on commodity parallel hardware such as multi-core CPU and GPU. We plan to investigate on data structures and parallel algorithm designs for the new hardware. The second level is to scale out the 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 of large-scale spatial data using existing software packages that are designed for uniprocessors and disk-resident systems based on a serial computing model.

3.2 Parallel Spatial Indexing
In this section, we will introduce our proposed designs on parallel spatial indexing. First, we will discuss our proposed spatial data layout that is expected to be efficient on both multi-core CPUs and GPUs. We will then introduce our parallel designs on three well-known spatial indexes, including 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 the building blocks for distributed computing to be described in Section 3.3.2 and Section 3.3.3.

3.2.1 Spatial Data Layout
Although several geometry representation formats such as Well-Known Text (WKT) have been adopted in many existing 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 propose novel spatial data layout designs for efficient in-memory geometry operations, which are cache friendly and are effective for data-parallel operations on both multi-core CPUs and GPUs.
Since OGC SFS\textsuperscript{12} has been widely adopted by the spatial database 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 \( \rightarrow \) feature \( \rightarrow \) ring \( \rightarrow \) vertex. 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 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 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 VPU and GPUs. Figure 11 gives an example of the SoA layout of a polygon collection.

![Figure 11 Spatial Data Layout Example](http://www.opengeospatial.org/)

In addition to the exact representation of geometry objects, approximate representation such as MBR is also important because it is widely adopted in spatial indexes. 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

\[\text{Figure 11 Spatial Data Layout Example}\]

\[\text{In addition to the exact representation of geometry objects, approximate representation such as MBR is also important because it is widely adopted in spatial indexes. 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}\]

\[\text{\textsuperscript{12} http://www.opengeospatial.org/}\]

22
MBR extraction procedure can be easily implemented by using a single *reduce_by_key* parallel primitive with the vertex array as one input and the MBR id array as another input to specify keys.

As detailed in the next few subsections, we also adopt SoA layouts for spatial index structures, including Grid-file, Quadtree and R-tree. BFS ordering of tree nodes is used for tree structures to improve parallelisms and cache locality on both CPUs and GPUs.

### 3.2.2 Parallel Grid-File based Indexing for MBRs

We first propose to develop an in-memory grid-file based spatial index on parallel platforms using data-parallel designs for both polylines and polygons that are represented as MBRs (Section 2.2). The designs are also applicable to points that can be considered as MBRs with zero extents. The data-parallel grid file index is designed for supporting efficient parallel spatial queries (this section) and spatial joins (Section 3.3). 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 propose a query strategy using binary search that is both efficient and requires no extra space. Third, for all the stages of index construction, our proposed data-parallel designs can be implemented using parallel primitives, which not only simplifies code complexity but also makes it portable across multiple parallel platforms.

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 by user. 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 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 end positions of rasterized MBRs and links grid cell ids and MBR ids. In our design, only grid cells that contain MBRs are stored for space efficiency. The middle part of Figure 12 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 calculate the output size for each

![Figure 12 Parallel Grid File based Indexing](image-url)
MBR. A scan is performed on the output size array in order to compute the starting position of each MBR. With the start 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 ids 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 12, 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 will also design parallel batched 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 12, we assume that {P1, P2} and {Q1, Q2} are the indexed MBRs and query MBRs, respectively. Without spatial index, the query needs to cross compare on all pairs, which is very expensive. To find the P-Q pairs that spatially intersect efficiently using the grid-file index, as illustrated in the right part of Figure 12, first, P1 and P2 are projected onto a grid space and indexed by array 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. Note that it is not necessary to sort and reduce QC or QQ to generate QC’ and QQ’ for efficiency purpose. 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 12, the query pair (3, 1) in QC and QQ array locates the corresponding cell in the index arrays (PC’ and PQ’) through binary search. As we mentioned previously, a PN array is used to link PC’ and PQ’, which is used for retrieving and pairing ids in PQ’ and QQ. As all the involved operations, i.e., sort, search and unique, can be efficiently parallelized in quite a few parallel libraries including Thrust, batched spatial query using grid-file indexing can be relatively easily implemented on multi-core CPUs and 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 limited memory, such as GPUs. This issue can be partially addressed by tuning grid cell sizes. Clearly using larger grid cells will have less MBR duplication but produce more false positives. Compared with the R-tree based spatial indexing, 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
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 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 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 13.

![Figure 13 Illustration of Linear R-Tree Node layout](image)

Compared with a previous work reported in [26] 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, R-trees 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 in-frequently updated, we will focus on bulk loading which allows simple and elegant implementations using parallel primitives.

3.2.3.2 Parallel R-tree construction

In this study, we propose to develop data parallel designs based on both low-x packing (used in [26]) and Sort-Tile-Recursive (STR) packing [24] to construct bulk-loaded R-trees.

For the low-x packing approach, the initialization step first sorts the original data (MBRs) is sorted by applying a linear ordering schema (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. The parallel design using parallel primitives is illustrated in the left part of Figure 14 where the R-tree is constructed by packing MBRs bottom-up. 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, same keys 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 as the sum of 1s for each child node.

We also propose the STR R-tree bulk loading algorithm 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 \textit{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 14 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 root 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 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 secondary key is \(y\)-coordinates and the primary 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). 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.

\begin{figure}[h]
\centering
\begin{minipage}{0.45\textwidth}
\textbf{Input:} fan-out \(d\); dataset \(D\)
\textbf{Output:} packed R-Tree
1. sort \(D\) using 1-D ordering (e.g. low-x)
2. for level \(\leftarrow\) num\(_{\text{level}}\) decrease to 1
3. \hspace{1em} if (level is last level)
4. \hspace{2em} reduce from original data \(D\)
5. \hspace{1em} else
6. \hspace{2em} reduce from lower level
\end{minipage}
\begin{minipage}{0.45\textwidth}
\textbf{Input:} fan-out \(d\); dataset \(D\)
\textbf{Output:} packed R-Tree
1. while (true)
2. \hspace{1em} if \((N \leq d)\)
3. \hspace{2em} root \(\leftarrow\) pack \(N\) MBRs
4. \hspace{2em} break;
5. \hspace{1em} else
6. \hspace{2em} sort\_by\_key on \(x\)-coordinates
7. \hspace{2em} sort\_by\_key on \(y\)-coordinates for each slice
8. \hspace{2em} reduce\_by\_key packed every \(d\) MBRs
9. \hspace{1em} \(N \leftarrow N/d\)
\end{minipage}
\caption{Parallel R-tree Bulk Loading}
\end{figure}

3.2.3.3 Parallel Batched Query on R-tree

After introducing the R-tree construction designs, we next propose our parallel designs for batched spatial range queries on R-trees. As we have introduced in Section 2.2.3, Luo et al [26]
proposed BFS based batched queries on GPUs, where several queries are assigned to a block and a queue is maintained for the whole block. With such a design, better intra-block load balance can be achieved and shared memory can be used to further speed up the query processing. The authors addressed queue overflow issue by adding another step to re-run finish overflowed queries respectively until completion. However, their design was tied to specific hardware (i.e., GPU) and may not be suitable for other parallel platforms. Meanwhile, the workload balance in [26] was limited to a block. In contrast, our design a global queue for all queries instead of many queues in [26], which produces 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 15 Parallel primitive based BFS batch query](image)

The left side of Figure 15 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 15. Two queries and their execution traces are colored in red and green, 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 query represents query results (F₁, G₁, I₂ and J₂).

We note that there are two potential issues in our design. First, the queue is maintained globally without considering specific hardware feature 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 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 3.4.1.1, 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 Point Data

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 propose to develop a parallel Quadtree indexing to index the points, which can be used to support spatial range queries and spatial joins. There are two steps in the proposed 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.

We propose the following data parallel design for generating leaf quadrants from point dataset and the idea is illustrated in Figure 16 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 ($K$) of points in a quadrant and the maximum level are set empirically by user.

![Figure 16 A Running Example to Illustrate the Process of Generating Point Quadrants](image)

The algorithm of generating point quadrants is listed in Figure 17. 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 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 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 \textit{len} array that has been used in R-trees. We propose a parallel primitive based design of constructing a complete Quadtree from its leaf quadrants, as listed in Figure 18. The input (\textit{Qs}) is a vector of leaf quadrants with their corresponding identifiers to the points and the output will be the constructed Quadtree (\textit{T}). We use \textit{z\_val}, \textit{lev} and \textit{p\_id} to represent Morton code, level and the corresponding point identifier respectively. At the beginning, Lines 1-2 are used to 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 \textit{current\_lev}, there are two major components. One component is directly coming from leaf quadrants. With the pre-generated level information at Line 3 and 4, we can easily locate leaf quadrants at \textit{current\_lev} and copy them to a temporary space (\textit{TempQs}). The other component

![Figure 17 Algorithm of Parallel Point Quadrant Generation](image)

![Figure 18 Parallel Quadtree Construction from Leaf Quadrants](image)
should come from reduction of lower level quadrants, in other words, the quadrants at current_lev+1. Those quadrants then are appended to TempQs. We needs to be emphasized that, to maintain the link between two levels, the first child position (fc) must be set appropriately. The last process is to copy the TempQs to the tree structure T (Line 12). After that, the iteration will continue on a higher level until the root of 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 Morton code. We thus skip the details of spatial query processing on Quadtrees for point datasets.

3.3 Parallel Spatial Join Design
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 propose to develop spatial join techniques for distributed environments to achieve scalability.

3.3.1 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 step spatial filtering 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 build 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 that can further improve pruning power and reduce number of tests of spatial predicates in the refinement step are preferable. Although the
spatial indexing and query processing techniques that we have proposed 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. The framework of our parallel spatial join technique on a single node is illustrated in Figure 19.

### 3.3.1.1 Parallel Spatial Filtering

Due to the close relationship between batched spatial query and spatial filtering, we first propose to compare the performance of the three parallel spatial indexing and query processing techniques introduced in Section 3.2 for spatial filtering. Here we assume one input dataset is indexed and the other input dataset is non-indexed. While query MBRs are unordered and non-indexed in batched spatial queries, for spatial joins, we propose to preprocess the MBRs derived from the non-indexed input dataset, including re-ordering and aggregation, to process spatial filtering more efficiently than simply treating the dataset as the query MBRs and apply the spatial query techniques.

Second, we would like to develop lightweight on-the-fly indexing techniques for spatial join involve points, such as point-in-polygon test based spatial join. Recent studies [32, 40] have shown that using non-hierarchical and simple spatial indexes on modern parallel hardware may produce better performance than using classic hierarchical 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 propose a lightweight on-the-fly indexing technique for the 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).

![Light-weighted Indexing for Point Dataset](image)

Assuming a grid-file has been created by indexing the input polygon/polyline dataset, the idea is to create a grid-file on-the-fly 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 layout 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 20. 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 proposed 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 can tolerate load unbalancing to a certain degree by using algorithms such as work stealing [27]. 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 propose to use cell-in-polygon test as an intermediate step between the spatial filtering step (based on grid cell matching) and spatial refinement (based on point-in-polygon test). The idea is illustrated in Figure 21. 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 performing the expensive point-in-polygon test 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 21, 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 paralleling the point-in-polygon test design to be described next. In addition to the point-in-polygon based spatial join, we will also explore the possibility of applying similar ideas to other types of spatial joins.

![Figure 21 Cell-to-polygon Relationship](image)

3.3.1.2 Parallel Refinement

The results of the filtering phase are candidate pairs that meet the spatial relationship but with false positives. Thus, a refinement phase is used to completely remove false positives and generate the final results. The refinement phase usually involves geometry 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 point-in-polygon test based spatial join.
Geometry operations have been well studied in computational geometry and implemented in several general-purpose geometry libraries such as JTS\(^{13}\) and GEOS\(^{14}\). 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 propose to develop a specialized geometry engine that is parallelized on both CPUs and GPUs based on our proposed spatial data layout introduced in Section 3.2.1. We propose to support 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 processing. We 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.

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 (thread in GPU and VPU lane in CPU). Using the classic ray-casting algorithm for point-in-polygon test \([16]\), 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 GPUs 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

\(^{13}\)http://www.vividsolutions.com/jts/JTSHome.htm

\(^{14}\)http://trac.osgeo.org/geos/
even O(1), we argue that the ray-casting algorithm does not require additional pre- and post-processing on the polygons, and the simplicity of its 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 is further illustrated in Figure 22. For GPUs, we assign a group of points to a GPU computing block, in which all points within the group performs 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, which 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.

3.3.2 Multi-Node Parallel Spatial Join
To perform spatial join on very large datasets, especially when the size of data exceeds the capacity of a single machine, we need to develop efficient spatial join techniques for multi-node computing environments, i.e., a cluster with multiple machines. We propose two distributed spatial join designs based on the characteristics of input datasets. If both datasets are very large and at a similar scale, we call the datasets symmetric. To process spatial joins on symmetric datasets (or symmetric spatial join), we propose to develop spatial partition based spatial join techniques, where data are divided based on a predefined spatial partition 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 two spatial join designs that take advantage of the state-of-the-art Big Data technologies, one is based on partition and the other is based on broadcast, after introducing several partition techniques.

### 3.3.2.1 Partition Techniques and Load Balancing

To efficiently process large-scale spatial datasets on multiple machines 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. The partition quality will directly impact the performance of distributed computing. As such, it is desirable to use a partition strategy can achieve good load balancing. We propose to investigate on two partition techniques, including non-spatial range partition and spatial partition. The non-spatial range partition, or range partition for short, divides a dataset into equal-sized partitions without considering spatial properties. Applying range partition requires no data re-ordering that is usually very expensive in distributed computing environments. Also, it can achieve data locality where data are accessed consecutively for a partition range. Range partition can be used in broadcast based spatial join which will be introduced in Section 3.3.2.3. On the other hand, spatial partition techniques take spatial properties of spatial datasets into consideration. We will introduce three partition strategies, i.e., Fixed-Grid Partition (FGP), Binary Split Partition (BSP) and Sort-Tile Partition (STP) [41]. They will be integrated in our partition based spatial join in Section 3.3.2.2. Examples are provided in Figure 23 to illustrate the three spatial partition techniques, respectively.

![Figure 23 Partition Examples](image)

Fixed-Grid Partition (FGP) is the most straightforward way of space decomposition, where the whole space is divided into grid partitions with equal size. This technique has been proposed and used in PBSM [35]. 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 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 the partition boundaries are determined by the 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 [4]. 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 [41]. Meanwhile, a parameter defines the maximum number of recursive level, which controls the number of generated 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 [24]. 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 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.

### 3.3.2.2 Spatial Partition based Spatial Join
We propose to develop 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 the shared-memory architectures within a single computing node, it is very expensive on shared-nothing cluster computers that involve data communication among distributed nodes. When designing the parallel spatial join technique 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 the stragglers (slow nodes). A good parallelization has to minimize the effects from stragglers. Therefore, the basic idea of our parallel 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 the spatial join task into small non-overlapping tasks that can run in parallel with low communication cost, 2) how to divide the spatial join task in a way that achieves better load balance. We introduce spatial partition based spatial join to address those challenges.

The 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 scheme 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. After that, each partition performs local spatial join 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- and intra-node parallelism. As intra-node parallelization has been discussed in previous sections, we focus on inter-node parallelization in this section. The overview of spatial partition based spatial join is illustrated in Figure 24. 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 dataset, 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. As discussed in previous works [36, 41], a good partition schema may result in better performance. In this proposal, we plan to develop data parallel designs and implementations of the three spatial partition techniques for our partition based spatial join on modern parallel and distributed platforms. To our knowledge, this has not been addressed in previous works.

After partitions are generated, both input datasets need to be shuffled based on the partition 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 a 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 propose to 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 propose to sort the combined results and remove duplication via a full scan on the results to reduce implementation complexity.

3.3.2.3 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 redundant 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. Consider a spatial join whose left side is a large point dataset and the right side is a moderately sized polyline or polygon dataset, we propose to broadcast the right side to all the partitions of the left side and perform spatial join locally. 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. 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 currently commodity computers.

The broadcast based spatial join in our design 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 broadcast. 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. Because the geometry objects of the small dataset are stored locally, the refinement phase can be performed without additional data transfer. 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 the 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 partition 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.

3.3.3 Proposed Prototype Systems
In order to demonstrate our proposed parallel designs introduced previously, we propose to develop two prototype systems based on Big Data technologies. The two prototype systems, called SpatialSpark and ISP, are based on Spark and Impala respectively.

3.3.3.1 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 propose to implement 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 Well-Known Text (WKT) format for storing geometry data in HDFS. As we are not using our proposed geometry engine at current phase in SpatialSpark, one of our future works is to replace JTS library with the new geometry engine for better performance.

For broadcast based spatial join, we take advantage of the broadcast mechanism in Spark, which can efficiently send a broadcast variable (which can be a dataset) to all nodes. JTS library is used to generate R-tree from the small input dataset and the geometries are associated with the leaf nodes. A broadcast variable is created from the generated R-tree, which can be accessed by all computing nodes. For the large dataset, each data item performs its local spatial join individually. We propose to use R-tree batched 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 pairs. As the output sizes of the queries may vary, we can use a flatMap primitive provided by Spark/Scala.

The spatial partition based spatial join is more complex than broadcast based spatial join. We propose to implement all three partition strategies introduced previously with both serial and parallel version variations on Spark for performance comparison purposes. 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 with a partition id using the built-in Scala `map` primitive. For the other two partition techniques, after an R-tree is constructed, each data item queries the R-tree in parallel to get the 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. After the shuffle phase, each partition contains two lists. Since the two lists are not indexed, we create an on-demand R-tree on one side and perform batched 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 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.

As our preliminary results reveal that JTS library is much slower than our proposed geometry engine, we propose to further improve SpatialSpark by replacing the JTS library with our proposed geometry engine introduced in Section 3.3.1.2. Since the geometry engine relies on our proposed data-parallel data structures which also support data-parallel indexing and query processing (Section 3.2), we also plan to implement selected spatial indexing techniques and use them to replace the R-tree indexing implemented in JTS. However, extending the data-parallel designs on a single-node to multi-node platforms can be challenging due to the intrinsic architectural differences. For instance, data access for different executors on a single node is much easier than in a multi-node distributed computing environment as the data may physically reside on another machine.

As SpatialSpark is very similar to existing packages such as Hadoop-GIS and SpatialHadoop, it is also desirable to evaluate the performance among the three platforms. We will conduct performance study for several real world spatial applications to gain insights. We also plan to test the scalability of SpatialSpark in the Cloud using both JTS library and our own SoA-based indexing and refinement engines on large datasets.

### 3.3.3.2 ISP

In addition to SpatialSpark, we propose to implement spatial joins on Impala which is another leading in-memory processing engine. The proposed implementation is called ISP, including a multi-core CPU version (ISP-MC) and a GPU version (ISP-GPU). Unlike Spark, Impala is implemented using C++, so 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. We believe ISP can fully take advantage of hardware accelerators. Figure 26 shows the preliminary architectural design of ISP.

First, we need to modify the Abstract Syntax Tree (AST) module of Impala frontend to support spatial query syntax. Second, we will represent 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 will integrate our single-node GPU-based spatial data management techniques with Impala to support large-scale spatial data processing on GPU-equipped clusters. We plan to implement broadcast based spatial join first due to its similarity with existing relational hash join implementation in Impala. For spatial partition based spatial
join, we will investigate the possibility of its implementation using existing infrastructure. Unlike Spark that provides convenient parallel primitives, Impala is an end-to-end system which makes it difficult to build custom applications.

We next present a design of the point-in-polygon test based spatial join accelerated by GPUs in ISP. First, we iteratively retrieve the geometry column of tuples of the “small” table and build a spatial index for all the tuples in it. In this study, we apply our proposed parallel R-tree indexing on the GPU. We note that retrieving the “small” table from HDFS can be efficiently done using multi-threaded I/O supported by Impala. Second, we iterate through all the row batches of the left side table that are assigned to an Impala instance sequentially to perform the 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. 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 of the left side table is transferred to GPUs for parallel query against the spatial index on the GPUs built in the first step. The query result is then transferred back to CPUs in the form of a vector of identifier pairs. Third, tuples of the left side table and right side 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 to be consumed by the 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). The process of a point-in-polygon test based spatial join using R-tree in ISP-GPU is illustrated in Figure 27.

We plan to investigate the scalability of ISP on multi-core CPU and GPU equipped clusters to accelerate the spatial join processing, including both filter and refinement phases. We propose to fully develop both ISP-CPU and ISP-GPU and investigate on the possibility of developing a more efficient distributed spatial join engine by learning the experiences on developing both the SpatialSpark and ISP prototypes.

3.4 Preliminary Results

3.4.1 Preliminary Results of Parallel Spatial Data Management on Single-Node

In this section, we will provide some preliminary results of single-node parallel spatial join based on our proposed designs, including R-tree based and grid file based implementations.

3.4.1.1 Data-Parallel R-tree Implementation

We have implemented data-parallel R-tree using parallel primitive library for both tree construction and batched query. Both the multi-core CPU and GPU parallel versions are realized for comparison purpose. To demonstrate performance, we use benchmark from [1] and compared the performance on both multi-core CPUs and GPUs. All experiments are performed on 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 installed.

The major component in R-tree construction that dominates the overall performance is the sorting phase. We 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 keys for sorting. The experiment results are given in the left chart of Figure 28, 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 28 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 that case, however, we still can process such large dataset using data partition techniques which are left for future work.
We have also implemented and evaluated the STR R-Tree bulk loading algorithm on multi-core CPUs and the GPUs. The results are given in the right chart of Figure 28 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. Based on the results shown in Figure 28, 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 3.2.3.2, 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.

![Figure 28 Performance of R-tree Construction](image)

![Figure 29 Speedups of GPU-based Implementations over Multi-Core CPU-based Implementations for Spatial Window Query Processing](image)

We also compare the performance of batched query processing on the GPU with multi-core CPU implementations. The multi-core CPU implementations utilize all available cores (8 cores in total) in the system based on OpenMP where each core is responsible for a single query. As can be seen from Figure 29, our GPU implementations have achieved about 10X speedup on average when compared with multi-core CPU implementations. For queries labeled with Q1 which uses small query windows, GPU implementations did 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.

### 3.4.1.2 Grid-file based Spatial Join

We have implemented grid-file based spatial join on both multi-core CPUs and GPUs. The design of grid-file based filtering uses the batch query processing technique that has been introduced in Section 3.2.2. Based on the efficient parallel implementation of grid-file based matching, we have implemented a point-in-polygon test based spatial join on GPUs. In this application, point quadrants are generated first using the Quadtree index as proposed 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.

We conduct experiments on 170 million pickup locations in 2009 from New York City taxi trip data, which are in format of latitude and longitude. The polygon data we use is the NYC Census 2000 dataset. There are more than 40 thousand census block polygons in NYC with more than 5 million vertices. The experiment platform is the same as introduced in Section 3.4.1.1. For comparison purposes, we have implemented the same spatial join using open source GIS packages, i.e., libspatialindex to index polygon data by building an R-tree, and, GDAL, which implicitly uses GEOS, to perform point-in-polygon test. The CPU implementation queries each point against the indexed polygons. If the point falls within any of the bounding boxes of polygons, the polygon identifiers will be returned for refinement. 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 a total end-to-end runtime of 11.2 seconds. In contrast, the serial CPU implementation takes 54,819 seconds (15.2 hours). As such, a significant speedup of 4,910X has been achieved. Note that we have not included the 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 point, polygon and auxiliary data are memory resident in our GPU implementation. In contrast, the open source

---


16 http://libspatialindex.github.com/

17 http://www.gdal.org/
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 misses. Second, in our GPU implementation, we have divided points into quadrants before we query against the polygons in the filtering 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 R-tree of the polygon dataset to leaf nodes, which is quite costly. While not tested in this study, we expect querying the bounding boxes of points in quadrants, instead of querying the points individually, can potentially improve the serial CPU implementation. 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 spatial join process, including generating point quadrants, filtering quadrant-polygon pairs and point-in-polygon test in computing blocks.

3.4.2 Preliminary Results on Multi-Node Parallel Spatial Join

3.4.2.1 SpatialSpark

We have implemented SpatialSpark for both broadcast based and spatial partition based spatial joins. 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 (PIP) and nearest-neighbor-search based spatial join (NNS). We have prepared a point dataset derived from pickup locations from NYC taxi trip data that has ~170 million points (termed as taxi dataset). To experiment the PIP spatial join, we use a polygon dataset from New York City census block data with about 40 thousand polygons (termed as nycb dataset). For the NNS spatial join, we use NYC street network (LIONf) dataset that has about 200K polylines (termed as lion dataset). All datasets are formatted and stored in text files in HDFS with geometries (points, polylines and polygons) in WKT format. The data volumes of the lion, nycb and lion datasets in HDFS are 6.9 GB, 18.7 MB and 29.0 MB, respectively. In addition to the taxi point dataset, we also prepare a subset of the GBIF species occurrence data. We extract about 10 million occurrences and formatted their geometry as WKT strings (termed as G10M dataset). The World Wide Fund global ecoregions consist of 14,458 polygons and 4,028,622 vertices (279 vertices per polygon on average), are used as the right side dataset (termed as wwf dataset) for PIP spatial joins and the experiment is labeled as G10M-wwf. The data volumes of the G10M and wwf datasets in HDFS are 12.9 GB and 149.8 MB, respectively.

We have evaluated the performance of the four experiments on a 10-node Amazon EC2 cluster and the results are plotted in Figure 30. All Amazon EC2 instances are equipped with 8 vCPU

---

(Intel Sandy Bridge 2.6 GHZ), 15 GB memory, 60 GB SSD and an NVIDIA GPU with 4 GB graphics memory and 1,536 CUDA cores. For taxi-lion experiments, we 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). Based on the results, SpatialSpark achieves very good scalability when number of instances increases. We plan to perform comparison study on existing work Hadoop-GIS and SpatialHadoop on the same set of experiments.

![SpatialSpark Performance](image.png)

**Figure 30 SpatialSpark Performance**

### 3.4.2.2 ISP

We conducted performance evaluation on two sets of experiments. The first experiment is joining taxi data (taxi, ~170 million points) with New York Census Block data (nycb, ~40 thousand polygons). The other experiment is joining species occurrence records of the Global Biodiversity Information Facility (GBIF) repository (snapshot 08/02/2012, termed as gbif) and the polygon dataset is from World Wild Fund (WWF) global ecological region data (termed as wwf). The number of point data items is about 375 million and the number of polygons is 14,458. We first report performance of ISP-MC and IPS-GPU on a workstation and then report the performance of the standalone versions of the two prototypes on the same machine to under system overhead. Finally, the performances on EC2 clusters are reported for discussions on scalability. The workstation is equipped with dual 8-core Intel Sandy Bridge 2.6 GHZ CPUs, 128 GB memory, 8 TB HDD and an NVIDIA GTX Titan GPU. The GTX Titan GPU has 6 GB GDDR5 memory and 2,668 CUDA cores. All Amazon instances (computing nodes) are the same as in Section 3.4.2.1. All machines are running CentOS 6.5 and Hadoop 2.5.0 from Cloudera CDH 5.2.0 with default settings. We empirically set the row batch capacity in Impala to accommodate a whole HDFS data block (64 MB by default).

The single-node performance for the two experiments is listed in the first two columns of Table 3. The runtimes are 96 seconds for taxi-nycb and 1,822 seconds for gbif-wwf for the ISP-GPU implementation. ISP-MC performs a little worse than ISP-GPU but 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 designs and parallel primitives based implementations, which are efficient on not only GPUs but also multi-core CPUs. Similar to the experiment reported in Section 3.4.2.1 on the spatial join on the taxi data, a serial implementation using libspatialindex for spatial indexing/filtering and GDAL (which uses GEOS for geometry operations) for spatial refinement can only achieve 138 points per second using a subset of GBIF data with 10 million points on an Intel Xeon processor (2.0 GHZ). 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 and the results are listed in the last two columns of Table 3. 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>
<thead>
<tr>
<th>Table 3 ISP Performance on Single Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISP-</td>
</tr>
<tr>
<td>GPU</td>
</tr>
<tr>
<td>taxi-nycb (s)</td>
</tr>
<tr>
<td>GBIF-WWF(s)</td>
</tr>
</tbody>
</table>

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 have extracted a subset of the WWF dataset with approximately 50 million points and we call the subset G50M and label the experiment as G50M-wwf. The scalability results for taxi-nycb and G50M-wwf experiments are plotted in Figure 31. 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 work load 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 the high-end workstation 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 31 Scalability Test Results of ISP-GPU and ISP-MC for taxi-nycb (left) and G50M-wwf (right) Experiments

4 Summary and Research Plan
In this proposal, we have presented preliminary designs and implementations of efficient and scalable spatial data management techniques on modern parallel and distributed platforms, including spatial indexing and spatial join query processing. The designs explored multiple levels of parallelisms, including coarse-grained parallelization on multiple nodes and fine-grained parallelization on a single node, and SIMD computing in VPU’s of multi-core CPUs and GPUs.
Preliminary experiment results have demonstrated orders of magnitude of speedups when compared with existing techniques that are designed for serial computing and disk-resident systems. The experiment results have also demonstrated good scalability of the multi-node spatial join techniques in Amazon EC2 cluster computers. Several performance bottlenecks that warrant further research. The thesis research plan is as listed below.

| February 2015 – April 2015 | • Develop batch query processing for the proposed spatial indexing techniques  
|                           | • Conduct performance study on SpatialSpark and ISP, and Compare with existing works such as Hadoop-GIS and SpatialHadoop  
|                           | • Design and implement a large scale spatial data processing framework by integrating existing components and leveraging experiences learnt from SpatialSpark and ISP developments |
| April 2015 – May 2015     | • Evaluate the impacts of different partition strategies for partition based spatial join and develop cost model to determine optimal partition schemas  
|                           | • Compare the impacts of utilizing different spatial indexes in spatial join processing |
| May 2015 – Sep 2015       | • Write and defend thesis |

5 Reference


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


