SJMR:Parallelizing Spatial Join with MapReduce on Clusters

Shubin Zhang[1,2], Jizhong Han[1], Zhiyong Liu[1], Kai Wang[1,2]
1. Institute of Computing Technology
2. Graduate University
Chinese Academy of Sciences
Beijing, China
{zhangshubin, hjz, zyliu, wangkai2008}@ict.ac.cn

Zhiyong Xu
Mathematics and Computer Science Department
Suffolk University
Boston, U.S.A
zxu@mcs.suffolk.edu

***Abstract*—MapReduce is a widely used parallel programming model and computing platform. With MapReduce, it is very easy to develop scalable parallel programs to process data-intensive applications on clusters of commodity machines. However, it does not directly support heterogeneous related data sets processing, which is common in operations like spatial joins. This paper presents SJMR (Spatial Join with MapReduce), a novel parallel algorithm to relieve the problem. The strategies include strip-based plane sweeping algorithm, tile-based spatial partitioning function and duplication avoidance technology. We evalauted the performance of SJMR algorithm in various situations with the real world data sets. It demonstrates the applicability of computing-intensive spatial applications with MapReduce on small scale clusters.**

I. Introduction

MapReduce [1] is a remarkable parallel programming model proposed by Google for large scale data processing on clusters of share-nothing commodity machines. Through a simple interface with two functions, map and reduce, this model facilitates easy parallel program implementation for many real-world tasks on clusters.

MapReduce has many advantages compared to other parallel processing models like PVM [2] and MPI [3]. It allows simple MapReduce programs to benefit from system mechanisms for communication, load balance, task scheduling and fault tolerance. MapReduce has been proved applicable and is effective for a wide range of applications.

As one of the most common spatial queries, spatial join combines two spatial data sets with a spatial relationship between the objects. Spatial join is widely used in applications such as spatial DBMS, robotics, game programming and VLSI designing, etc. Thus, efficient spatial join processing is extremely important since its execution time is super-linear with the object numbers of the involving data sets. It might be a good idea to implement spatial join with MapReduce on clusters. By taking advantage of parallel computing, we can greatly reduce the processing time for such a computing-intensive operation.

However, it is difficult to parallelize spatial join with MapReduce. First, MapReduce focuses mainly on processing homogeneous data sets, while spatial join has to deal with two heterogeneous data sets [4]. Second, the spatial join predicates are complex and time-consuming, and spatial objects are generally larger and more complex than words or URL strings in common MapReduce applications. Third, spatial partitioning operation produces considerable data duplications which make it difficult to build a parallel spatial join operator.

In this paper, we provide a detailed discussion on parallelizing spatial join with MapReduce on clusters. This paper makes three contributions.

First, Spatial Join with MapReduce (SJMR), a novel parallel spatial join algorithm without spatial indexes, is presented. The algorithm homogenizes and splits inputs into disjoint partitions evenly at Map stage with a tile-based spatial partitioning function. SJMR joins the partitions at Reduce stage with a strip-based plane sweeping algorithm that can be considered as the equivalent spatial algorithm of hash-merge and sort-merge. Using MapReduce parallel programming model, SJMR makes the parallelizing of spatial join simple, flexible, scalable, and robust.

Second, a simple and effective tile-based duplication avoidance method is introduced. This mechanism makes it possible to build a parallel spatial join algorithm on clusters of share-nothing machines.

Third, the performance of SJMR is compared with the traditional PPBSM (Parallel Partition Based Spatial–Merge join) algorithm [5], [6], [7]. We evaluate their performance based on the actual implementations of these two algorithms on Hadoop [8], an open source implementation of Google's GFS [9] and MapReduce. Using real world data sets from the TIGER/Line [10] files, we examine the behaviors of the SJMR algorithm in various situations. It shows that MapReduce is applicable for computing-intensive spatial applications and small scale clusters.

The remainder of this paper is organized as follows. Section II introduces MapReduce and related works in spatial join area. Section III describes the SJMR algorithm. Section IV introduces the tile-based duplication avoidance method. The performance evaluation is presented in Section V. Finally, Section VI concludes our paper.

II. Background and related work

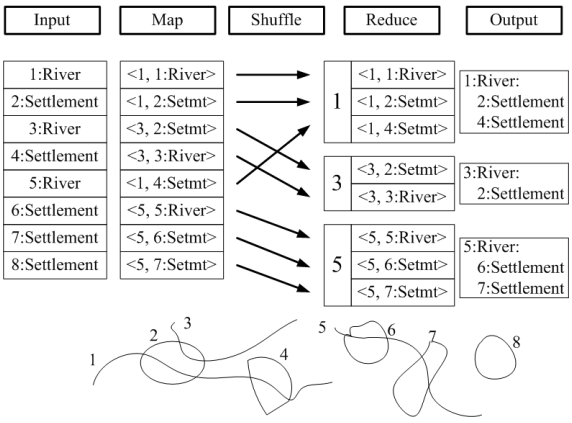
A. MapReduce

MapReduce is a programming model and computing platform suited for parallel computation [1]. In MapReduce, a program consists of a map function and a reduce function which are user-defined. The input data format is application-related, and is specified by the user. The output is a set of $< key, value >$ pairs. The MapReduce signatures are listed below:

$$\begin{aligned} map &: (k_1, v_1) \rightarrow [(k_2, v_2)] \\ reduce &: (k_2, [v_2]) \rightarrow [k_3, v_3] \end{aligned} \quad (1)$$

As shown in the signature, the map function applies user-defined logic on every input key/value pair (k_1, v_1) and transforms it into a list of intermediate key/value pairs $[(k_2, v_2)]$. Then the reduce function applies user-defined logic to all intermediate values $[v_2]$ associated with the same k_2 and produces a list of final output key/value pairs $[k_3, v_3]$.

Fig 1 shows an example of MapReduce application, calculating the settlements passed through by each river. The curves in the Fig 1 denote rivers, and polygons denote settlements.




```
//input: spatial objects (rivers and settlements)
//       key=spatial object ID,
//       value=spatial object property

map(Integer key, String value):
    for each value:
    EmitIntermediate(river ID, "ID:property");

//intermediate: key=river ID,
//              value=spatial object ID:property

reduce(Integer key, Iterator values):
//including the Shuffle step
    for each value sharing the same key:
    Emit(AsString(value));

//output: settlements passed through by the same
//        river
```

Fig. 1. MapReduce example

The MapReduce framework accomplishes the parallelization of map and reduce operations by dividing the responsibilities of those operations among many tasks. Input data sets are partitioned among multiple mappers, which are tasks responsible for applying the map function. Mappers retrieve their input data from a distributed file system and write tuples emitted to a group of files on local disk. These files are then accessed by another group of tasks called reducers.

Each mapper produces a single file for a reducer, containing the tuples that produce the reducer's key when hashed. Once a reducer has received its files from all mappers, it sorts and merges the files by keys and reduces each key in turn, outputting the resulting tuples to files on the distributed file system.

MapReduce provides many benefits over other parallel processing models. It allows simple data processing programs to benefit from MapReduce implementation for communication, load balance, fault tolerance, resource allocation, job startup, and file distribution.

Introduced by Google, MapReduce is used in a wide range of applications, such as web access log stats, web link-graph reversal, machine learning, statistical machine translation and so on. There are usually hundreds of computers used for such a computation. MapReduce is not only implemented for large-scale cluster computing, but can also be used in multi-core and other environments. Phoenix [11] is an implementation of MapReduce model to program multi-core chips as well as shared-memory multiprocessors. MapReduce has also been implemented on GPUs, such as Mars [12]. This paper demonstrates that MapReduce can also achieve good performance in computation-intensive spatial applications and small-scale clusters.

B. Spatial Join

Given two sets of multidimensional objects in Euclidean space, a spatial join query can discover all pairs of objects satisfying a given spatial relation, such as intersection [13]. For example, a spatial join answers such queries like finding all the roads spanning any hydrography, given a road table and a hydrography table.

Since the representation of a spatial object can be very large and complex, spatial join typically operate in two steps:

Filter Step: An approximation of each spatial object, such as its minimum bounding rectangle (MBR), is used to check with the spatial predication, eliminating tuples that cannot be parts of the result. This step produces candidates which are a superset of the actual result.

Refinement Step: Each candidate pair of the last step is examined to check whether its spatial properties satisfy the spatial predicate. A CPU-intensive computational geometry algorithm is generally used.

Numerous algorithms have been proposed to execute the filter step of a spatial join. Most recent algorithms were based on using spatial indices on both input data sets [14], [15], [16]. But the situation is different when the input data sets are the results of some other spatial queries or the input data sets are

dynamically redistributed just as MapReduce. Neither input data set will have a spatial index. As a possible solution, an index can be built on one or both data sets online, and then the index-based spatial join techniques can be used. However, in MapReduce data is redistributed dynamically, other techniques that do not depend on time-expensive indexes might be faster. The key to most of these techniques lies in partitioning the data sets so that the partitions are small enough to fit in internal memory, then internal memory techniques can be used.

For the filter step, parallel techniques extending the synchronized hierarchical traversal approach have been used for indexed data [17], and techniques extending the grid partitioning method have been used for unindexed data [6], [18], [19]. These techniques assumed a shared-nothing architecture, although some algorithms were extended to use shared-memory architectures to improve performance [17], [19]. Most of these methods have shown a near linear speed-up when filtering with more processors. However, they depend on good load balancing strategies. An algorithm which used a hypercube architecture to join two indexed data sets was also discussed [20].

For unindexed data, Zhou [19] adapted the variation of the grid partitioning method that physically created the grid cells. In their approach, the random data was evenly divided among *n* processors, which then partitioned the data using the same tiling scheme. Then, with the sizes of the grid cells known, a single processor sequentially determined the merging of grid cells into *N* partitions using a *Z*-order merge, creating an even-load distribution amongst the processors. Next, each processor was assigned a partition and the data was redistributed appropriately. Finally, each processor filtered the data using a sequential spatial join filtering technique. All in all, to gain even load distribution, this method introduced sequential partitioning course. Both Patel [6] and Luo [18] investigated a similar approach. Rather than physically tiling the data, they both used the virtual tiling method, relying on a hash function to distribute the data evenly.

For parallel refinement, each processor could refine the candidate pairs it produces. Zhou [19] pointed out that it was difficult without much selectivity information to balance the number of candidates produced by each processor at the filtering stage. They argued that an efficient approach was to use one processor to sort the candidates into a linear order, and then assign the candidates to each processor in a round-robin fashion. Just like its grid partitioning method, Zhou introduced inefficient sequential method into the parallel environment again.

III. Spatial join with MapReduce on clusters

This section describes a novel parallel algorithm, named Spatial Join with MapReduce (SJMR), for evaluating spatial join with MapReduce on clusters. For the sake of concreteness, the two input data sets of spatial join are mapped into relations *R* and *S*. It's also assumed that each tuple has a unique identifier, referred to as *OID*.

The SJMR algorithm operates in the following two stages.

Map stage: First of all, every tuple of *R* and *S* is homogenized and extracted some key attributes, such as *OID*, *MBR* and *spatial property*, etc. According to the novel tile-based partitioning method introduced later, every tuple is then mapped into one or more partitions. After this stage, every tuple generates one or more (k_2, v_2) pairs. k_2 corresponds to partition number, v_2 includes *OID*, *MBR*, *spatial property* and *tiling information*.

SJMR assumes that each partition corresponds to one Reduce task. Every partition is filtered and refined at one Reduce task. Let *P* be the number of partitions and Reduce tasks, then the value range of k_2 is [0, . . ., P-1].

Reduce stage: This stage is composed of *filter step* and *refinement step*. At the filter step, a strip-based plane sweeping technique is used to join the partitions. The result of this step is a set of *OID* pairs $< T_R, T_S >$. For each pair, the *MBR* of T_R overlaps with that of T_S. Since two non-overlapping spatial features can have overlapping *MBR*s, the filter step generally produces a superset of the join result. In the refinement step, the *R* and *S* tuples pointed to by T_R and T_S are fetched from disk, and their spatial properties are examined to determine whether the join predicate is actually satisfied.

The following is the detailed description of SJMR algorithm.

A. Determining the partition number

The minimum number of partitions (Reduce tasks) is determined by several factors. It can be estimated as following. Let $\|R\|$ and $\|S\|$ represent the cardinalities of *R* and *S*, and duplication coefficient be *p*. Let *M* represents the size of available main memory in each node, and $Size_{kp}$ denotes the size of a key-pointer element (which includes *MBR*, *OID* and *tiling information* of a tuple). Since the plane sweeping algorithm used in merging the partitions requires the partitions, R_i^{kp} and S_i^{kp}, to fit entirely in memory, the minimum number of partitions is calculated as:

$$P = [(\|R\| + \|S\|) \times (1 + p) \times Size_{kp}/M] \qquad (2)$$

B. Map stage

The tuples of *R* and *S* are distributed on the data nodes according to HDFS (Hadoop's Distributed FileSystem) distributing strategy. So the first stage of SJMR needs to redistribute the tuples of *R* and *S* into different Reduce tasks according to *spatial partitioning function* (*SPF*). The goal is to distribute the tuples so that each Reduce task performs roughly equal work and the distribution will not affect the validity of results. This is done with the help of *SPF* proposed below.

1) Homogenizing step: MapReduce focuses mainly on processing homogeneous data sets. However, in most cases, a spatial join needs to merge two heterogeneous data sources. So the first task of Map stage is to homogenize the data sources.

In this step, a *data-source tag* is inserted into every tuple, with the method of adding a data-source attribute into v_2. Homogenized data sets accordingly have four common attributes: *OID*, *MBR*, *spatial property* and *data-source*.

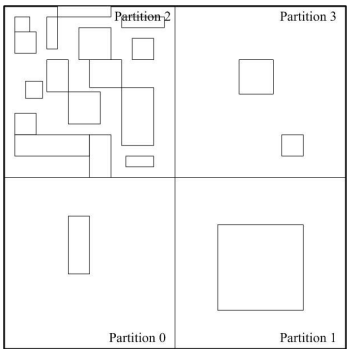


Fig. 2. Spatial data skew

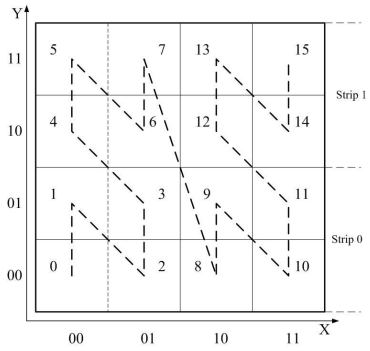


Fig. 3. *Z* curve coding

2) Spatial splitting step: The *universe* is defined as the minimum rectangle that covers all tuples of *R* and *S*. With the presence of non-uniform distribution, splitting the universe into *P* partitions directly will produce partitions that have large differences in size. As shown in Fig 2, most of the tuples are in the top left corner, and *SPF* will map all those tuples to *partition 2*. *Partitions 0*, *1* and *3* will have very few tuples. This will lead to uneven memory and CPU requirements for different Reduce tasks.

To remedy data skew situation in parallel relational joins, [21] brought forward virtual processor round robin partitioning. A similar partitioning function was proposed for parallel spatial database [22], but in this function, tile number was equal to partition number. [5] adapted a variation of grid partitioning method which split universe into N_T same-sized tiles ($N_T \geq P$), each tile was coded line-based and hashed to *P* partitions.

We believe that the design space of *SPF* has three axes: *tile number*, *tile coding method*, and *tile-to-partition mapping scheme*. SJMR adopts a tile-based partitioning method which splits universe into N_T same-sized tiles ($N_T \gg P$) just like [5], but has a different tile coding method and tile-to-partition mapping scheme.

Decomposing the universe into smaller tiles makes it easier to produce a more uniform partition distribution. However, spatial objects that span tiles from multiple partitions have to be replicated in all those partitions, thereby increasing the replication overhead.

For the tile-to-partition mapping scheme, either *round robin* or *hashing* method could be used.

With tile coding method, we use spatial clustering technology to make partition distribution more uniform. There is no natural order in multi-dimensional space, but the mapping from multi-dimension to one-dimension should keep distance relationship fixed. To this end, a lot of space-filling curves were put forward, but none of them can completely meet that goal. *Z curve* and *Hilbert curve* are the best choices. Fig 3 gives an example of *Z curve*.

To explore all alternatives in constructing *SPF*, we conducted experiments with a data set extracted from the TIGER/-Line files [10], which is used to represent roads of California. Details of the data set are described in Section V. In Fig 4 and Fig 5, the *line-based* tile coding method collaborates with *round robin* or *hashing* mapping scheme, while *Z curve* and *Hilbert curve* tile coding methods make use of *round robin* mapping scheme.

Fig 4 shows the effect of different *SPF*s to coefficient of variation. The graph uses the *coefficient of variation* of tuple numbers in different partitions as metric. A perfect *SPF* would assign equivalent tuples to each partition, and consequently, would have coefficient of variation to be 0. In Fig 4, the numbers that follow tile coding methods are partition numbers.

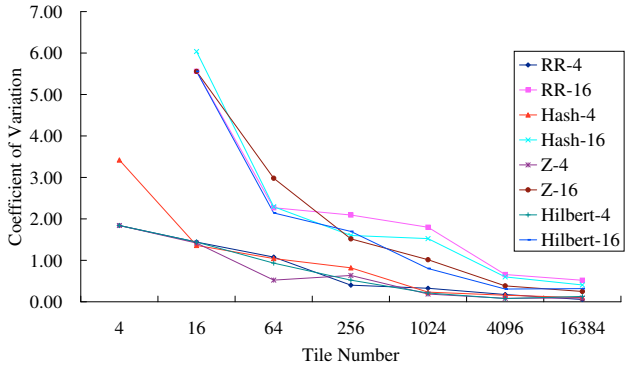


Fig. 4. Coefficient of variation

From Fig 4, the following observations can be found. First of all, for *Z curve* and *Hilbert curve* coding methods combined with *round robin* mapping scheme, in most cases their coefficients of variation are superior to that of *line-based* coding combined with *hash* mapping proposed by [5], which is superior to *line-based* coding combined with *round robin* mapping. Second, performances of all *SPF*s improve as tile number increases, because intensive regions are divided up. Third, for a given tile number, the *SPF*s yield a more uniform distribution with a smaller partition number. This is because the distribution of tiles covering ”dense” regions is better with a smaller number of partitions.

Fig 5 measures the *replication overhead*–the increase of tuple number by partitioning–for various tile numbers. It shows that, for the Tiger data set, the replication overhead is highly modest even for a very large tile number. And just

as analyzed above, with the increase of tile number, replication overhead is raising.

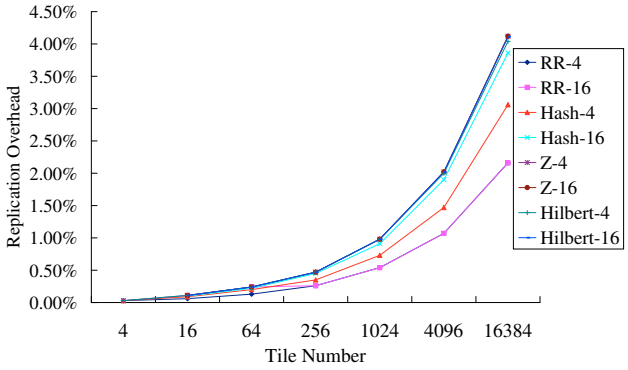


Fig. 5. Replication overhead

According to the experimental results, *Z curve* tile coding method combining with *round robin* mapping schema is chosen as *SPF* in SJMR. Since *Z curve* is easier to compute, and has equivalent performance with *Hilbert curve*.

The *SPF* of SJMR works as follows. First, the universe is divided regularly into N_T tiles, where $N_T \gg P$. And each tile is numbered from 0 to $N_T - 1$ according to *Z* curve, and mapped to a partition p $(0 \leq p \leq P-1)$ with a *round robin* scheme. Then for each tuple, *SPF* checks its *MBR* to determine all the tiles with which the *MBR* overlaps and redistributes this tuple by setting its k_2 as the numbers of partitions those tiles belonging to. So if *MBR* of the tuple overlaps with several partitions, it will be redistribute to all of them.

The duplication avoidance and filtering step of Reduce stage need the *tiling information* of every tuple, thus the *tiling information* needs to be saved as another attribute of every tuple. So intermediate value v_2 of every tuple have five common attributes: *OID*, *MBR*, *spatial property*, *data-source* and *tiling information*.

As a result, tuples from different data sets with the same key k_2 will be grouped in the same Reduce task. User-defined logic can extract data-sources from values to identify their origins, and the entries from different sources can be merged.

C. Reduce stage

With the advancement of remote sensing technology, the description of spatial objects will be more precise. It is not acceptable to compute on spatial properties of tuples directly. In the area of spatial DBMSs, *MBR* is the most common approximation of spatial objects. As a result, spatial join is carried out in two steps: *filter step* and *refinement step*, just as we do at the Reduce stage.

1) Filter step: The filter step of Reduce stage begins by reading intermediate value v_2 of every tuple belonging to the same partition k_2. According to the data-source attribute, key-pointer element of each value of v_2 is appended to one of two temporary relations, R^{kp} and S^{kp}, in memory, and the other information of the tuple is appended to one of two temporary relations, R^T and S^T, on disk.

The main goal of the filter step is to "pair" tuples from the same partition so that their *MBR*s could overlap. The problem then simplifies to finding all *MBR*s in R^{kp} that intersect with some *MBR*s in S^{kp}. Rectangle intersection has been extensively studied in the computational geometry field [23]. Given two sets of rectangles, when both the sets fit entirely in main memory, efficient plane sweeping techniques exist for reporting all pairs of intersecting rectangles between the two sets. Now, it's ensured by Map stage that both R^{kp} and S^{kp} fit in memory, so a plane sweeping algorithm can be used to find all pairs of key-pointer elements in R^{kp} and S^{kp} that have overlapping *MBR*s. For such "matching" key-pointer element pairs, the *OID* information is extracted and added to the output of this step.

To make full use of location information of tuples got by splitting, SJMR adopts a novel *strip-based plane sweeping* method. In this method, every partition is divided equally into several strips according to tiles. The strips are side by side and parallel with x-axis. Then every strip is filtered with a plane sweeping algorithm. Every tuple is allocated into the strips according its tiling information. For example, the universe is divided into 2 strips in Fig 3.

```
procedure StripPlaneSweep(partitionR, partitionS)
begin
    MRs ← StripAndSort(partitionR)
    MSs ← StripAndSort(partitionS)
    foreach corresponding MR ∈ MRs and MS ∈ MSs do
        repeat until MR and MS are empty
            Let mr = Head(MR) and ms = Head(MS)
            if mr^x_min < ms^x_min
                Remove mr from MR
                Scan MS from the current position
                and report all rectangles that
                intersect mr (with reference tile method)
                Stop when the current rectangle
                r ∈ MS satisfies
                mr^x_max < r^r_min
            else
                Remove ms from MS
                Scan MR from the current position
                and report all rectangles that
                intersect ms(with reference tile method)
                Stop when the current rectangle
                r ∈ MR satisfies
                ms^x_max < r^x_min
            endif
        end
    done
end
```

Fig. 6. Strip-based plane sweeping algorithm

The strip-based plane sweeping algorithm is listed in Fig 6. The key parameter of strip-based plane sweeping algorithm is strip number. The effect of different strip numbers is shown in Section V.

2) Refinement step: The filter result is a temporary relation whose tuples have the form $< OID_R, OID_S >$, which means *MBR* of OID_R overlapping with *MBR* of OID_S.

The refinement step examines the spatial properties of OID_R and OID_S to see if the tuples actually satisfy the join condition. To avoid random seeks in fetching *R* and *S* tuples saved in R^T and S^T on disk, a strategy used in [5], [24] is employed. First, the *OID* pairs are sorted using OID_R as the primary sort key and OID_S as the secondary sort key. Next, as many *R* tuples in R^T as that can fit in memory are read from disk along with the corresponding array of $< OID_R, OID_S >$ pairs. The OID_R part of this array is "swizzled" to point to the *R* tuples in memory, and then the array is sorted on OID_S (this makes the accesses to S^T sequential). The *S* tuples in S^T are then read sequentially into memory, and the spatial properties of the *R* and *S* tuples in memory are checked to determine whether they satisfy the join condition.

To avoid generating duplicates in results, SJMR adopts a tile-based duplication avoidance technology. With this technology, spatial join algorithm need not to sort the results and remove duplicates sequentially, and could be parallelized effectively. This technology is introduced below.

IV. Duplication avoidance technology

In SJMR, the join result of two tuples T_R and T_S may be generated several times in two ways:

1. If both T_R and T_S are replicated to several partitions at Map stage, duplicates will be generated at the Reduce tasks where both of them are replicated.

2. At a specific Reduce task, if both T_R and T_S are replicated in several strips, duplicates will be generated in strips where both of T_R and T_S are replicated.

There are two ways to remove duplicates from outcomes: duplication avoidance and duplication elimination. Duplication elimination method needs to sort refine results and remove duplicates. This method could only proceed when Reduce stage has finished completely, so its cost is high.

Rather than using a duplication elimination operator at the end of SJMR, the better way is to avoid generating duplicates online at each Reduce task. So the filter step is modified with a simple test applied when the rectangles are checked for intersection. The technique, named *reference tile method*, is an improvement to *reference point method* [25]. For each pair of T_R and T_S, if both of them are replicated at several Reduce tasks, they are only joined at one of the tasks where both of them are replicated. According to the tiling information of every tuple, reference tile method calculates the *common smallest tile* of the two tuples. The result pair is reported only if the *common smallest tile* is within the current partition and strip.

Fig 7 shows the situation in which the universe is partitioned into sixteen tiles. The common smallest tile of T_R and T_S is *Tile3*, so they only be joined at the Reduce task and the strip where *Tile3* lies in.

Compared with the reference point method, the biggest advantages of our approach are smaller amount of calculation by making full use of tiling information acquired at Map stage, and perfect combination with strip-based plane sweeping algorithm.

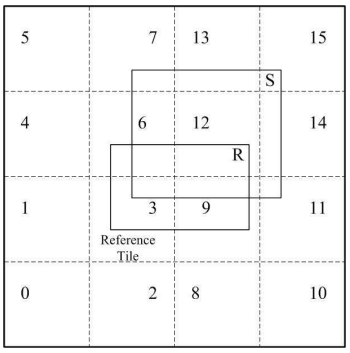


Fig. 7. Reference tile method

V. Performance evaluation

A. Experiment environment and data sets

The Experiments were performed with Hadoop 0.18.1 running on DELL Power Edge SC430 Servers each consisting of one Intel Pentium 4 2.80 GHz processor, with 1GB main memory, and 80GB SATA disk. RedHat AS4.4 with kernel 2.6.9 operating system and ext3 file system were running on each server.

The experimental results were acquired on 1-node, 2-node, 4-node, and 8-node configurations. Each node acted as a tasktracker and datanode, with an independent server as jobtracker and namenode. 2 Map tasks and 2 Reduce tasks could be executed simultaneously on each node.

The classic PPBSM algorithm was implemented using MapReduce for performance comparison. Because PBSM performs best among all the traditional spatial join algorithms when neither input has a pre-existing spatial index, which is the same as MapReduce.

Two data sets from TIGER/Line files were used. One includes the road information of California, and the other includes hydrographies in California. Below is the statistic information of the two data sets.

TABLE I
California TIGER data

Data Set	Object Number	Size	Average Point Number
Road	2092079	529MB	3.87
Hydrography	373950	135MB	10.77

B. Impact of strip number

To evaluate the impact of strip number on strip-based plane sweeping algorithm, the algorithm (except refine step) is implemented on one single node. With the data sets of road and hydrography, the experiment results are shown in Fig 8.

Fig 8 shows that strip number has obvious impact on the strip-based plane sweeping algorithm. With the increase of strip number, strip-based plane sweeping algorithm gets better

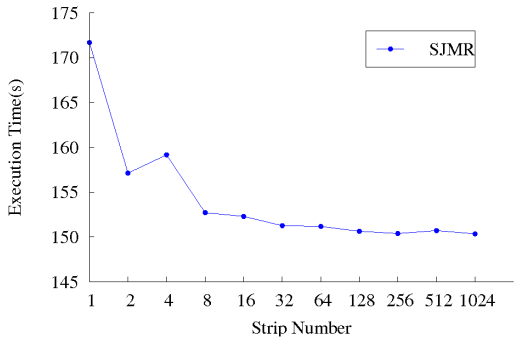


Fig. 8. Impact of strip number

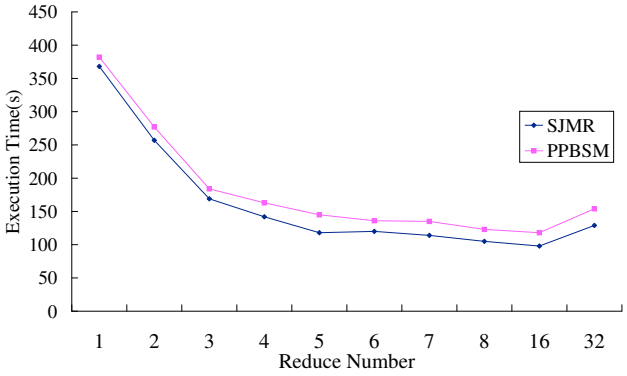


Fig. 10. Performance comparison of SJMR and PPBSM (8 nodes)

performance. A good performance is achieved when strip number is 8, so in strip-based plane sweeping algorithm of SJMR the strip number is set to be 8.

C. Impact of node number

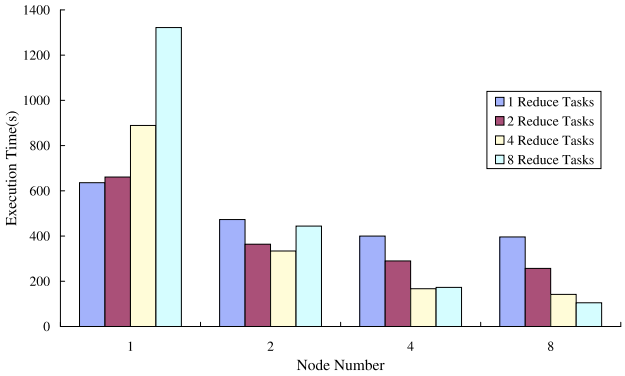


Fig. 9. Impact of node number

Fig 9 shows the impact of node numbers and Reduce task numbers on SJMR performance. From this figure, it could be concluded that the performance of the SJMR algorithm has direct relationship with the numbers of nodes and Reduce tasks. Firstly, with the increasing of node number, performance of SJMR improves obviously. It could be deduced that better performance would be obtained with more nodes by analyzing SJMR. Secondly, for a certain node number, *N*, at the time of $(P \leq 2N)$, performance of SJMR improves with the increase of Reduce task number. But when $P > 2N$, performance of SJMR recedes with the increase of Reduce task number. This is because 2 Reduce tasks could be executed simultaneously at every tasktracker. When $P > 2N$, the Reduce tasks could not be executed simultaneously in one cycle.

D. Performance comparison of SJMR and PPBSM

Because PPBSM eliminates duplications after spatial join, two MapReduce jobs are used to implement PPBSM on Hadoop. Just like SJMR, the Map stage of the first job is to split and homogenize the data sources, but the Reduce stage will complete the filter step and refine step just like PBSM algorithm [5]. The Map stage of the second job is just to split *OID* pairs from the first job and to send on the *OID* pairs with OID_R as k_2, OID_S as v_2. And the Reduce stage will output each $< k_2, v_2 >$ pair only once.

As shown in Fig 10, the performance of both SJMR and PPBSM improves with the increase of Reduce task number *P*, when $P \leq 2N$, since more finer-grained Reduce tasks could be executed at the same time with the same workload. But when $P > 2N$, both performance begin to deteriorate, because the Reduce tasks could not be completed in one cycle. The performance of PPBSM is always worse than SJMR for about 20 seconds. When Reduce task number is 16, performance of PPBSM is about 20.4% worse than that of SJMR. This is because duplicates in PPBSM need to be removed with an independent MapReduce job which costs about 20 seconds. It also shows that though a duplication avoidance test is added in SJMR, with strip-based sweeping algorithm, performance of SJMR is comparable with that of PPBSM without duplication elimination.

E. Performance comparison of SJMR and SJMR-LargeMem

Due to memory size limitation on every node, Reduce stage of SJMR needs to save spatial property of every tuple to the temporary files of T_R and T_S on local disk. In order to evaluate the impact of disk-write operations, Reduce task number and memory size of each node are increased to make the memory size large enough to filter and refine all in memory without writing operations. What's more, the SJMR algorithm is revised to make it not to write spatial property of every tuple to the temporary files on disk. The revised algorithm is named as SJMR-LargeMem.

With the same Reduce task number of 8, the performance of SJMR and SJMR-LargeMem with different node numbers are compared in Fig 11. As is shown in this figure, in most cases the SJMR algorithm even has a better performance than SJMR-LargeMem. This is because the latter needs to convert spatial property of every tuple from string to spatial object. While SJMR only reads in the spatial property of each tuple belonging to the result of the filtering step and converts them into spatial object, a lot of converting time is saved. And with the increase of node number, performance of both algorithms improve, because more Reduce tasks could be

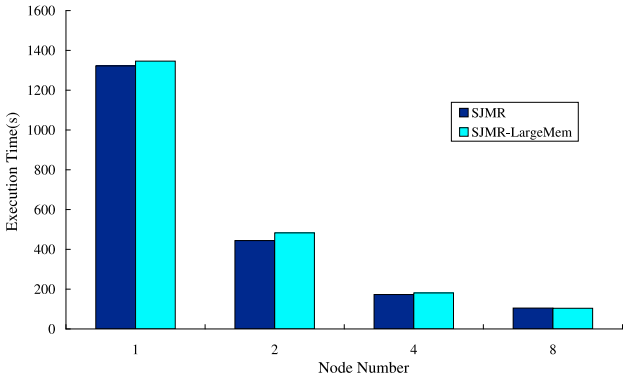


Fig. 11. Performance comparison of SJMR and SJMR-LargeMem (8 reduce tasks)

executed simultaneously.

VI. Conclusions

In this paper, we describe how spatial join can be effectively implemented and accelerated with MapReduce on clusters. The algorithm designed is named as SJMR (Spatial Join with MapReduce). As far as we know, SJMR is the first parallel spatial join algorithm optimized for MapReduce, which allows spatial join to be processed using MapReduce platform on clusters of commodity machines.

SJMR algorithm splits the inputs into disjoint partitions with a spatial partitioning function at the Map stage, and merges every partition with a strip-based plane sweeping algorithm and a tile-based duplicate avoidance method at the Reduce stage. The strategies of SJMR can also be used in other parallel environments, especially where neither of the inputs has spatial index.

Using real world data sets, this paper also presents a performance evaluation comparing SJMR with the PPBSM algorithm implemented with MapReduce. The results demonstrate the feasibility and efficiency of the SJMR algorithm, and show that MapReduce is applicable in computation-intensive spatial applications and small scale clusters.

VII. Acknowledgement

This research is partially supported by the National Basic Research Program of China (973 Program, 2004CB318202 and 2007CB310805),National Natural Science Foundation of China (Grant No. 60752001), the Project of Research & Development Plan of High Technology in China (863 Project, 2009AA12Z226). The work is also supported in part by the National Science Foundation Grant CNS-0509207. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107, 2008.

[2] A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*, MIT Press Cambridge, MA, USA, 1995.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1999.

[4] R. Pike, "Interpreting the data: Parallel analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[5] J.M. Patel and D.J. DeWitt, "Partition based spatial-merge join," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 1996, pp. 259–270.

[6] J.M. Patel and D.J. DeWitt, "Clone join and shadow join: two parallel spatial join algorithms," in *Proceedings of the 8th ACM international symposium on Advances in geographic information systems*. ACM New York, NY, USA, 2000, pp. 54–61.

[7] J. Patel, J.B. Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellmann, et al., "Building a scaleable geo-spatial DBMS: technology, implementation, and evaluation," *ACM SIGMOD Record*, vol. 26, no. 2, pp. 336–347, 1997.

[8] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," *Wiki at http://lucene. apache. org/hadoop*.

[9] S. Ghemawat, H. Gobioff, and S.T. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[10] UC Bureau, "Census 2007 Tiger/Line data," 2007.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," *hpca*, vol. 0, pp. 13–24, 2007.

[12] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, New York, NY, USA, 2008, pp. 260–269, ACM.

[13] E.H. Jacox and H. Samet, "Spatial join techniques," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 1, 2007.

[14] T. Brinkhoff, H.P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using R-trees," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 1993, pp. 237–246.

[15] O. Gunther and FAW Ulm, "Efficient computation of spatial joins," in *Proceedings of the Ninth International Conference on Data Engineering*, 1993, pp. 50–59.

[16] E.G. Hoel and H. Samet, "Benchmarking Spatial Join Operations with Spatial Output," in *Proceedings of The International Conference on Very Large Data Bases*. Institute of Electrical & Electronics Engineers (IEEE), 1995, pp. 606–618.

[17] T. Brinkhoff, H.P. Kriegel, and B. Seeger, "Parallel processing of spatial joins using R-trees," in *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pp. 258–265.

[18] G. Luo, JF Naughton, and CJ Ellmann, "A non-blocking parallel spatial join algorithm," in *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, 2002, pp. 697–705.

[19] X. Zhou, D.J. Abel, and D. Truffet, "Data Partitioning for Parallel Spatial Join Processing," *GeoInformatica*, vol. 2, no. 2, pp. 175–204, 1998.

[20] E.G. Hoel and H. Samet, "Data-Parallel Spatial Join Algorithms," in *ICPP '94: Proceedings of the 1994 International Conference on Parallel Processing*, 1994, vol. 3.

[21] D.J. DeWitt, J.F. Naughton, D.A. Schneider, and S. Seshadri, "Practical Skew Handling in Parallel Joins," in *Proceedings of the International Conference on Very Large Data Bases*. Institute of Eletrical & Electronics Engineers (IEEE), 1992, pp. 27–27.

[22] K.L. Tan and J.X. Yu, "A Performance Study of Declustering Strategies for Parallel Spatial Databases," *Lecture Notes in Computer Science*, pp. 157–157, 1995.

[23] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer, 1985.

[24] P. Valduriez, "Join Indices," *ACM Transactions on Database Systems*, vol. 12, no. 2, pp. 218–246, 1987.

[25] J.P. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing," in *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, 2000, pp. 535–546.