Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce

Ablimit Aji[1] Fusheng Wang[2] Hoang Vo[1] Rubao Lee[3] Qiaoling Liu[1] Xiaodong Zhang[3] Joel Saltz[2]

[1]Department of Mathematics and Computer Science, Emory University
[2]Department of Biomedical Informatics, Emory University
[3]Department of Computer Science and Engineering, The Ohio State University

ABSTRACT

Support of high performance queries on large volumes of spatial data becomes increasingly important in many application domains, including geospatial problems in numerous fields, location based services, and emerging scientific applications that are increasingly data- and compute-intensive. The emergence of massive scale spatial data is due to the proliferation of cost effective and ubiquitous positioning technologies, development of high resolution imaging technologies, and contribution from a large number of community users. There are two major challenges for managing and querying massive spatial data to support spatial queries: the explosion of spatial data, and the high computational complexity of spatial queries. In this paper, we present *Hadoop-GIS* – a scalable and high performance spatial data warehousing system for running large scale spatial queries on Hadoop. Hadoop-GIS supports multiple types of spatial queries on MapReduce through spatial partitioning, customizable spatial query engine RESQUE, implicit parallel spatial query execution on MapReduce, and effective methods for amending query results through handling boundary objects. Hadoop-GIS utilizes global partition indexing and customizable on demand local spatial indexing to achieve efficient query processing. Hadoop-GIS is integrated into Hive to support declarative spatial queries with an integrated architecture. Our experiments have demonstrated the high efficiency of Hadoop-GIS on query response and high scalability to run on commodity clusters. Our comparative experiments have showed that performance of Hadoop-GIS is on par with parallel SDBMS and outperforms SDBMS for compute-intensive queries. Hadoop-GIS is available as a set of library for processing spatial queries, and as an integrated software package in Hive.

1. INTRODUCTION

The proliferation of cost effective and ubiquitous positioning technologies has enabled capturing spatially oriented data at an unprecedented scale and rate. Collaborative spatial data collection efforts, such as OpenStreetMap [6], further accelerate the generation of massive spatial information from community users. Analyzing large amounts of spatial data to derive values and guide decision making have become essential to business success and scientific discoveries. For example, Location Based Social Networks (LBSNs) are utilizing large amounts of user location information to provide geo-marketing and recommendation services. Social scientists are relying on such data to study dynamics of social systems and understand human behavior.

The rapid growth of spatial data is driven by not only industrial applications, but also emerging scientific applications that are increasingly data- and compute- intensive. With the rapid improvement of data acquisition technologies such as high-resolution tissue slide scanners and remote sensing instruments, it has become more efficient to capture extremely large spatial data to support scientific research. For example, digital pathology imaging has become an emerging field in the past decade, where examination of high resolution images of tissue specimens enables novel, more effective ways of screening for disease, classifying disease states, understanding disease progression and evaluating the efficacy of therapeutic strategies. Pathology image analysis offers a means of rapidly carrying out quantitative, reproducible measurements of micro-anatomical features in high-resolution pathology images and large image datasets. Regions of micro-anatomic objects (millions per image) such as nuclei and cells are computed through image segmentation algorithms, represented with their boundaries, and image features are extracted from these objects. Exploring the results of such analysis involves complex queries such as spatial cross-matching, overlay of multiple sets of spatial objects, spatial proximity computations between objects, and queries for global spatial pattern discovery. These queries often involve billions of spatial objects and heavy geometric computations.

A major requirement for the data intensive spatial applications is fast query response which requires a scalable architecture that can query spatial data on a large scale. Another requirement is to support queries on a cost effective architecture such as commodity clusters or cloud environments. Meanwhile, scientific researchers and application developers often prefer expressive query languages or interfaces to express complex queries with ease, without worrying about how queries are translated, optimized and executed. With the rapid improvement of instrument resolutions, increased accuracy of data analysis methods, and the massive scale of observed data, complex spatial queries have become increasingly compute- and data-intensive due to following challenges.

The Big Data Challenge. High resolution microscopy images from high resolution digital slide scanners provide rich information about spatial objects and their associated features. For example, whole-slide images (WSI) made by scanning microscope slides at diagnostic resolution are very large: A typical WSI contains 100,000x100,000 pixels. One image may contain millions of objects, and hundreds of image features can be extracted for each object. A study may involve hundreds or thousands of images ob-

tained from a large cohort of subjects. For large scale interrelated analysis, there may be dozens of algorithms - with varying parameters - to generate many different result sets to be compared and consolidated. Thus, derived data from images of a single study is often in the scale of tens of terabytes. A moderate-size hospital can routinely generate thousands of whole slide images per day, which can lead to several terabytes of derived analytical results per day, and petabytes of data can be easily created within a year. For the OpenStreetMap project, there have been more than 600,000 registered contributors, and user contributed data is increasing continuously.

High Computation Complexity. Most spatial queries involve geometric computations which are often compute-intensive. Geometric computation is not only used for computing measurements or generating new spatial objects, but also as logical operations for topology relationships. While spatial filtering through minimum bounding rectangles (MBRs) can be accelerated through spatial access methods, spatial refinements such as polygon intersection verification are highly expensive operations. For example, spatial join queries such as spatial cross-matching or overlaying multiple sets of spatial objects on an image or map can be very expensive to process.

The large amounts of data coupled with compute-intensive nature of spatial queries require a scalable and efficient solution. A potential approach for scaling out spatial queries is through a parallel DBMS. In the past, we have developed and deployed a parallel spatial database solution – *PAIS* [31, 30, 7]. However, this approach is highly expensive on software licensing and dedicated hardware, and requires sophisticated tuning and maintenance efforts [26].

Recently, MapReduce based systems have emerged as a scalable and cost effective solution for massively parallel data processing. Hadoop, the open source implementation of MapReduce, has been successfully applied in large scale internet services to support big data analytics. Declarative query interfaces such as Hive [29], Pig [19], and Scope [17] have brought the large scale data analysis one step closer to the common users by providing high level, easy to use programming abstractions to MapReduce. In practice, Hive is widely adopted as a scalable data warehousing solution in many enterprises, including Facebook. Recently we have developed a system *YSmart* [22], a correlation aware SQL to MapReduce translator for optimized queries, and have integrated it into Hive.

However, most of these MapReduce based systems either lack spatial query processing capabilities or have limited spatial query support. While the MapReduce model fits nicely with large scale problems through key-based partitioning, spatial queries and analytics are intrinsically complex and difficult to fit into the model due to its multi-dimensional nature [**?**]. There are two major problems to handle for spatial partitioning: spatial data skew problem and boundary object problem. The first problem could lead to load imbalance of tasks in distributed systems and long response time, and the second problem could lead to incorrect query results if not handled properly. In addition, spatial query methods have to be adapted so that they can be mapped into partition based query processing framework while preserving the correct query semantics. Spatial queries are also intrinsically complex which often rely on effective access methods to reduce search space and alleviate high cost of geometric computations. Thus, there is a significant step required on adapting and redesigning spatial query methods to take advantage of the MapReduce computing infrastructure.

We have developed *Hadoop-GIS* [5] – a spatial data warehousing system over MapReduce. The goal of the system is to deliver a scalable, efficient, expressive spatial querying system to support analytical queries on large scale spatial data, and to provide a feasible solution that can be afforded for daily operations. Hadoop-GIS provides a framework on parallelizing multiple types of spatial queries and having the query pipelines mapped onto MapReduce. Hadoop-GIS provides spatial data partitioning to achieve task parallelization, an indexing-driven spatial query engine to process various types of spatial queries, implicit query parallelization through MapReduce, and boundary handling to generate correct results. By integrating the framework with Hive, Hadoop-GIS provides an expressive spatial query language by extending HiveQL [**?**] with spatial constructs, and automates spatial query translation and execution. Hadoop-GIS supports fundamental spatial queries such as point, containment, join, and complex queries such as spatial cross-matching (large scale spatial join) and nearest neighbor queries. Structured feature queries are also supported through Hive and fully integrated with spatial queries.

The rest of the paper is organized as follows. We first present an architectural overview of Hadoop-GIS in Section 2. The spatial query engine is discussed in Section 3, MapReduce based spatial query processing is presented in Section 4, boundary object handling for spatial queries is discussed in Section 5, integration of spatial queries into Hive is discussed in Section 6, performance study is discussed in Section 7, which followed by related work and conclusion.

2. OVERVIEW

2.1 Query Cases

There are five major categories of queries: i) feature aggregation queries (non-spatial queries), for example, queries for finding mean values of attributes or distribution of attributes; ii) fundamental spatial queries, including point based queries, containment queries and spatial joins; iii) complex spatial queries, including spatial cross-matching or overlay (large scale spatial join) and nearest neighbor queries; iv) integrated spatial and feature queries, for example, feature aggregation queries in a selected spatial regions; and v) global spatial pattern queries, for example, queries on finding high density regions, or queries to find directional patterns of spatial objects. In this paper, we mainly focus on a subset of cost-intensive queries which are commonly used in spatial warehousing applications. Support of multiway join queries and nearest neighbor queries are discussed in our previous work [10], and we are planning to study global spatial pattern queries in our future work.

In particular, spatial cross-matching/overlay problem involves identifying and comparing objects belonging to different observations or analyses. Cross-matching in the domain of sky survey aims at performing one-to-one matches in order to combine physical properties or study the temporal evolution of the source [24]. Here spatial cross-matching refers to finding spatial objects that overlap or intersect each other [32]. For example, in pathology imaging, spatial cross-matching is often used to compare and evaluate image segmentation algorithm results, iteratively develop high quality image analysis algorithms, and consolidate multiple analysis results from different approaches to generate more confident results. Spatial cross-matching can also support spatial overlays for combining information for massive spatial objects between multiple layers or sources of spatial data, such as remote sensing datasets from different satellites. Spatial cross-matching can also be used to find temporal changes of maps between time snapshots.

2.2 Traditional Methods for Spatial Queries

Traditional spatial database management systems (SDBMSs) have been used for managing and querying spatial data, through extended spatial capabilities on top of ORDBMS. These systems often have major limitations on managing and querying spatial data

at massive scale, although parallel RDBMS architectures [**?**] can be used to achieve scalability. Parallel SDBMSs tend to reduce the I/O bottleneck through partitioning of data on multiple parallel disks and are not optimized for computationally intensive operations such as geometric computations. Furthermore, parallel SDBMS architecture often lacks effective spatial partitioning mechanism to balance data and task loads across database partitions, and does not inherently support a way to handle boundary crossing objects. The high data loading overhead is another major bottleneck for SDBMS based solutions [26]. Our experiments show that loading the results from a single whole slide image into a SDBMS can take a few minutes to dozens of minutes. Scaling out spatial queries through a parallel database infrastructure is studied in our previous work [30, 31], but the approach is highly expensive and requires sophisticated tuning for optimal performance.

2.3 Overview of Methods

The main goal of Hadoop-GIS is to develop a highly scalable, cost-effective, efficient and expressive integrated spatial query processing system for data- and compute-intensive spatial applications, that can take advantage of MapReduce running on commodity clusters. To realize such system, it is essential to identify time consuming spatial query components, break them down into small tasks, and process these tasks in parallel. An intuitive approach is to spatially partition the data into buckets (or tiles), and process these buckets in parallel. Thus, generated tiles will become the unit for query processing. The query processing problem then becomes the problem on designing querying methods that can run on these tiles independently, while preserving the correct query semantics. In MapReduce environment, we propose the following steps on running a typical spatial query, as shown in Algorithm 1.

In step A, we perform effective space partitioning to generate tiles. In step B, spatial objects are assigned tile UIDs, merged and stored into HDFS. Step C is for pre-processing queries, which could be queries that perform global index based filtering, queries that do not need to run in tile based query processing framework. Step D performs tile based spatial query processing independently, which are parallelized through MapReduce. Step E provides handling of boundary objects (if needed), which can run as another MapReduce job. Step F does post-query processing, for example, joining spatial query results with feature tables, which could be another MapReduce job. Step G does data aggregation of final results, and final results are output into HDFS. Next we briefly discuss the architectural components of Hadoop-GIS (Hive^{SP}) as shown in Figure 1, including data partitioning, data storage, query language and query translation, and query engine. The query engine consists of index building, query processing and boundary handling on top of Hadoop.

2.4 Data Partitioning

Spatial data partitioning is an essential initial step to define, generate and represent partitioned data. There are two major considerations for spatial data partitioning. The first consideration is to avoid high density partitioned tiles. This is mainly due to potential high data skew in the spatial dataset, which could cause load imbalance among workers in a cluster environment. Another consideration is to handle boundary intersecting objects properly. As MapReduce provides its own job scheduling for balancing tasks, the load imbalance problem can be partially alleviated at the task scheduling level. Therefore, for spatial data partitioning, we mainly focus on breaking high density tiles into smaller ones, and take a recursive partitioning approach. For boundary intersecting objects, we take the multiple assignment based approach in which objects

Algorithm 1: Typical workflow of spatial query processing on MapReduce

A. Data/space partitioning;
B. Data storage of partitioned data on HDFS;
C. Pre-query processing (optional);
D. **for** $tile$ **in** $input\_collection$ **do**
    Index building for objects in the tile;
    Tile based spatial querying processing;
E. Boundary object handling;
F. Post-query processing (optional);
G. Data aggregation;
H. Result storage on HDFS;

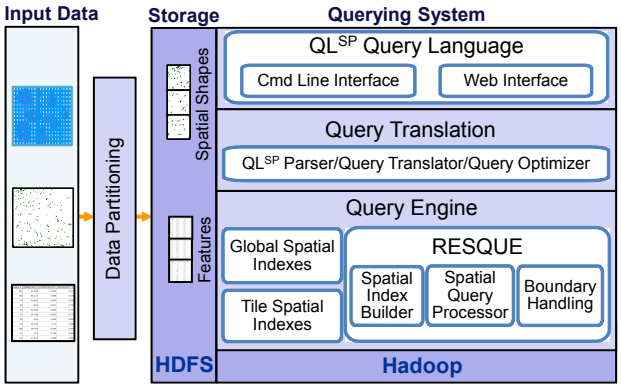


Figure 1: Architecture overview of Hadoop-GIS (Hive^{SP})

are replicated and assigned to each intersecting tile, followed by a post-processing step for remedying query results (section 5).

2.5 Real-time Spatial Query Engine

A fundamental component we aim to provide is a standalone spatial query engine with such requirements: i) is generic enough to support a variety of spatial queries and can be extended; ii) can be easily parallelized on clusters with decoupled spatial query processing and (implicit) parallelization; and iii) can leverage existing indexing and querying methods. Porting a spatial database engine for such purpose is not feasible, due to its tight integration with RDBMS engine and complexity on setup and optimization. We develop a Real-time Spatial Query Engine (RESQUE) to support spatial query processing, as shown in the architecture in Figure 1. RESQUE takes advantage of global tile indexes and local indexes created on demand to support efficient spatial queries. Besides, RESQUE is fully optimized, supports data compression, and comes with very low overhead on data loading. This makes RESQUE a highly efficient spatial query engine compared to a traditional SDBMS engine. RESQUE is compiled as a shared library which can be easily deployed in a cluster environment. Hadoop-GIS takes advantage of spatial access methods for query processing with two approaches. At the higher level, Hadoop-GIS creates global region based spatial indexes of partitioned tiles for HDFS file split filtering. As a result, for many spatial queries such as containment queries, we can efficiently filter most irrelevant tiles through this global region index. The global region index is small and can be stored in a binary format in HDFS and shared across cluster nodes through Hadoop distributed cache mechanism. At the tile level, RESQUE supports an indexing on demand approach by building tile based spatial indexes on the fly, mainly for query processing

purpose, and storing index files in the main memory. Since the tile size is relatively small, index building on a single tile is very fast and it greatly enhances spatial query processing performance. Our experiments show that, with increasing speed of CPU, indexing building overhead is a very small fraction of compute- and data-intensive spatial queries such as cross-matching.

2.6 MapReduce Based Parallel Query Execution

Instead of using explicit spatial query parallelization as summarized in [15], we take an implicit parallelization approach by leveraging MapReduce. This will much simplify the development and management of query jobs on clusters. As data is spatially partitioned, the tile name or UID forms the key for MapReduce, and identifying spatial objects of tiles can be performed in mapping phase. Depending on the query complexity, spatial queries can be implemented as map functions, reduce functions or combination of both. Based on the query types, different query pipelines are executed in MapReduce. As many spatial queries involve high complexity geometric computations, query parallelization through MapReduce can significantly reduce query response time.

2.7 Boundary Object Handling

In the past, two approaches were proposed to handle boundary objects in a parallel query processing scenario, namely Multiple Assignment and Multiple Matching [23, 36]. In multiple assignment, the partitioning step replicates boundary crossing objects and assigns them to multiple tiles. In multiple matching, partitioning step assigns an boundary crossing object to a single tile, but the object may appear in multiple tile pairs for spatial joins. While the multiple matching approach avoids storage overhead, a single tile may have to be read multiple times for query processing, which could incur increase in both computation and I/O. The multiple assignment approach is simple to implement and fits nicely with the MapReduce programming model. For example, spatial join on tiles with multiple assignment based partitioning can be corrected by eliminating duplicated object pairs from the query result. This can be implemented as another MapReduce job with some small overhead (Section 5).

2.8 Declarative Queries

We aim to provide a declarative spatial query language on top of MapReduce. The language inherits major operators and functions from ISO SQL/MM Spatial, which are also implemented by major SDBMSs. We also extend it with more complex pattern queries and spatial partitioning constructs to support parallel query processing in MapReduce. Major spatial operations include spatial query operators, spatial functions, and spatial data types.

2.9 Integration with Hive

To support feature queries with a declarative query language, we use Hive, which provides a SQL like language on top of MapReduce. We extend Hive with spatial query support by extending HiveQL with spatial constructs, spatial query translation and execution, with integration of the spatial query engine into Hive query engine (Figure 1). The spatial indexing aware query optimization will take advantage of RESQUE for efficient spatial query support in Hive.

3. REAL-TIME SPATIAL QUERY ENGINE

To support high performance spatial queries, we first build a standalone spatial querying engine RESQUE with following capabilities: i) effective spatial access methods to support diverse spatial query types; ii) efficient spatial operators and measurement functions to provide geometric computations; iii) query pipelines to support diverse spatial queries with optimal access methods; and iv) to be able to run with decoupled spatial processing in a distributed computing environment. We have adopted a set of open source spatial and geometric computation libraries to support diverse access methods and geometric computations, including SpatialIndex [8], GEOS [**?**], and build additional ones such as Hibert R-Tree [21]. Diverse spatial query pipelines are developed to support different types of queries based on the query type and data characteristics.

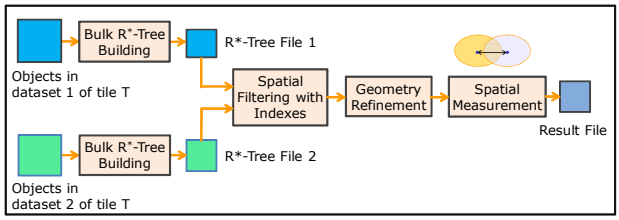


Figure 2: An example spatial join workflow in RESQUE

3.1 Indexed Spatial Query Support

One essential requirement for spatial queries is fast response. This is important for both exploratory studies on massive amounts of spatial data with a large space of parameters and algorithms, and decision making in enterprise or healthcare applications. Using spatial index to support spatial queries is a common practice for most SDBMS systems. However, the mismatch between the large data blocks in HDFS for batch processing and the page based random access of spatial indexes makes it difficult to pre-store spatial indexes on HDFS and retrieve it later for queries. While some effort has been made on this [35], the approaches are not flexible and the pre-generated indexes might not be suitable to support dynamic spatial query types. To support indexing based spatial queries, we combine two approaches: i) global spatial indexes for regions and tiles; and ii) on demand indexing for objects in tiles.

Global region indexes are small due to their large data granularity. They can be pre-generated and stored in a binary format in HDFS and shared across cluster nodes through Hadoop distributed cache mechanism. Global region indexing will facilitate region level data filtering and helps improve query performance. For an example, to process a point or window query we can quickly lookup the global index to identify the tiles that are relevant for the query range.

We propose an approach on building indexes on-demand combined with data partitioning to process spatial queries. Our extensive profiling of spatial queries shows that index building on modern hardware is not a major bottleneck in large scale spatial query processing. Using dynamically built local indexes for objects in tiles could efficiently support spatial queries with minimal index building overhead. To provide page based spatial index search, the built spatial indexes are stored in the main memory for query processing. In rare cases where not enough main memory is available for in memory index processing, secondary storage can be utilized for index storage. Our tests show the indexing building time using RESQUE with R*-Tree based join takes a very small fraction of the overall response time (Section 7.4).

3.2 Spatial Query Workflows in RESQUE

Based on the indexing methods above, we are able to create multiple spatial query pipelines. Next we discuss workflows for spatial join, spatial containment in detail, and nearest neighbor queries is

discussed in our previous work [10] which will be skipped here.

Spatial Join Workflow. Next we show a workflow of a spatial join (Figure 2), where two datasets from a tile T are joined to find intersecting polygon objects. The SQL expression of this query is discussed in Section 4.2. Bulk spatial index building is performed on each dataset to generate index files – here we use R*-Trees [12]. Hilbert R-Tree can also be used when the objects are in regular shapes and relatively homogenous distribution. The R*-Tree files are stored in the main memory and contain MBRs in their interior nodes and polygons in their leaf nodes, and will be used for further query processing. The spatial filtering component performs MBR based spatial join filtering with the two R*-Trees, and refinement on the spatial join condition is further performed on the polygon pairs through geometric computations. Much like predicate pushdown in traditional database query optimization, spatial measurement step is also performed on intersected polygon pairs to calculate results required, such as overlap area ratio for each pair of intersecting markups. Other spatial join operators such as *overlaps* and *touches* can also be processed similarly.

Index Optimization. As data and indexes are read-only and no further update is needed, bulk-loading techniques [13] are used. To minimize the number of pages, the page utilization ratio is also set to 100%. In addition, we provide compression to reduce leaf node shape sizes through compact chain code based representation: instead of representing the full coordinates for each x,y coordinate, we use offset from neighboring point to represent the coordinates. The simple chain code compression approach can save 40% space for the pathology imaging use case.

Spatial Refinement and Measurement. For each pair of markup polygons whose MBRs intersect, precise geometry computation algorithm is used to check whether the two markup polygons actually intersect. Spatial refinement based on geometric computation often dominates the query execution cost in data-intensive spatial queries, and could be accelerated through GPU based approach [32]. We are planning to integrate GPU based spatial functions into MapReduce in our future work.

Spatial Containment Workflow. Spatial containment queries have a slightly different workflow. The spatial containment query range may span only a single tile or multiple tiles. Thus, an initial step will be to identify the list of intersecting tiles by looking up the global tile index, which could filter a large number of tiles. The tiles whose MBRs intersect with the query range will then be further processed, where only a single index is used in the spatial filtering phase. For an extreme case where the containing shape is small and lies within a tile, only a single tile is identified for creating the index. For a point query – given a point, find the containing objects, only a single tile is needed and it has similar workflow as the small containment query.

4. MAPREDUCE BASED SPATIAL QUERY PROCESSING

RESQUE provides a core query engine to support spatial queries, which enables us to develop high performance large scale spatial query processing based on MapReduce framework. Our approach is based on spatial data partitioning, tile based spatial query processing with MapReduce, and result normalization for tile boundary objects.

4.1 Spatial Data Partitioning and Storage

Spatial data partitioning serves two major purposes. First, it provides two-dimensional data partitioning and generates a set of tiles, which become a processing unit for querying tasks. A large set of such tasks can be processed in parallel without data dependency or communication requirement. Therefore, spatial partitioning provides not only data partitioning but also computational parallelization. Last, spatial data partitioning could be critical to mitigate spatial data skew. Data skew is a common issue in spatial applications. For example, with a fixed grid partitioning of images into tiles with size of 4Kx4K, the largest count of objects in a tile is over 20K objects, compared to the average count of 4,291. For OpenStreetMap dataset, by partitioning the space into 1000x1000 fixed grids, the average count of objects per tile is 993, but the largest count of objects in a tile is 794,429. If there is a parallel spatial query processing based on tiles, such large skewed tile could significantly increase the response time due to the straggling tiles.

As MapReduce provides its own job scheduling for balancing tasks, for spatial data partitioning, we mainly focus on breaking high density regions into small ones, and take a recursive partitioning approach. We either assume the input is a pre-partitioned tileset with fixed grid size, which is commonly used for imaging analysis applications, or pre-generate fixed grid based tileset if no partitioning exists. We count the number of objects in each tile, and sort them based on the counts. We define a threshold C_{max} as the maximal count of objects allowed in a tile. We pick all tiles with object counts larger than C_{max}, and split each of them into two equal half-sized tiles based on an optimal direction: x or y. A direction is considered optimal if the split along that direction generates a new tile with object count below the threshold, or the two new tiles are more balanced. This process is repeated until all tiles have counts below than C_{max}.

After partitioning, each object in a tile is assigned a corresponding tile UID. The MBRs of tiles are maintained in a global spatial index. Note that in the same spatial universe, there could be multiple types of objects with different granularity, e.g., cells versus blood vessels, each dataset of a different type will have its own separate partitioning. If there are multiple datasets of the same type in the same space, e.g., two segmentation results of different algorithms, partitioning is considered together for all these datasets based on combined object counts.

For data staging into HDFS, we merge all tiles into large files instead of storing each tile as a separate file, as the file size from each tile could be small, e.g., a few MBs, which are not suitable to be stored directly into HDFS. This is due to the nature of HDFS, which is optimized for large data blocks (default block size 64MB) for batch processing. Large number of small files leads to deteriorated performance for MapReduce due to following reasons. First, each file block consumes certain amount of main memory on the cluster namenode and this directly compromises cluster scalability and disaster recoverability. Second, in the Map phase, the large number of blocks for small files leads to "large number of small map tasks" which has significant task-startup overhead.

For objects across tile boundaries, we take the multiple assignment approach, where an object intersects with the tile boundary will be assigned multiple times to all the intersecting tiles [23]. Consequently, such boundary objects may participate in multiple query tasks which could lead to incorrect query results. Therefore, the query results are normalized through an additional boundary handling process in which results are rectified (Section 5). While this method will incur redundancy on object storage, the ratio of boundary objects is usually small (within a few percent in our use cases).

4.2 Spatial Join with MapReduce

As spatial join is among the most commonly used and costly queries, next we discuss how to map spatial join queries into MapRe-

duce computing model. We first show an example spatial join query for spatial cross-matching in SQL, as shown in Figure 3. This query finds all intersecting polygon pairs between two result sets generated from an image by two different methods, and compute the overlap ratios (intersection-to-union ratios) and centroid distances of the pairs. The table *markup_polygon* represents the boundary as *polygon*, algorithm UID as *algorithm_uid*. The SQL syntax comes with spatial extensions such as spatial relationship operator *ST_INTERSECTS*, spatial object operators *ST_INTERSECTION* and *ST_UNION*, and spatial measurement functions *ST_CENTROID*, *ST_DISTANCE*, and *ST_AREA*.

```
1: SELECT
2:  ST_AREA(ST_INTERSECTION(ta.polygon,tb.polygon))/
3:    ST_AREA(ST_UNION(ta.polygon,tb.polygon)) AS ratio,
4:  ST_DISTANCE(ST_CENTROID(tb.polygon),
5:    ST_CENTROID(ta.polygon)) AS distance,
6: FROM  markup_polygon ta JOIN markup_polygon tb ON
8:    ST_INTERSECTS(ta.polygon, tb.polygon) = TRUE
9: WHERE ta.algrithm_uid='A1' AND tb.algrithm_uid='A2' ;
```

Figure 3: An example spatial join (cross-matching) query

For simplicity, we first present how to process the spatial join above with MapReduce, by ignoring boundary objects, and then we come back to the boundary handling in Section 5. A MapReduce program for spatial join query (Figure 3) will have similar structure as a regular relational join operation, but with all the spatial part executed by invoking RESQUE engine within the program. According to the equal-join condition, the program uses the Standard Repartition Algorithm [14] to execute the query. Based on the MapReduce structure, the program has three main steps: i) In the map phase, the input table is scanned, and the WHERE condition is evaluated on each record. Only those records that satisfy the WHERE condition are emitted with *tile_uid* is as key. ii) In the shuffle phase, all records with the same *tile_uid* would be sorted and prepared for the reducer operation. iii) In the reduce phase, the tile based spatial join operation is performed on the input records and the spatial operations are executed by invoking the RESQUE engine (Section 3).

The workflow of the map function is shown in the map function in Algorithm 2. Each record in the table is converted into the map function input key/value pair (k, v), where k is the byte offset of the line in the file, and v is the record. Inside the map function, if the record can satisfy the select condition, then an intermediate key/value pair is generated. The intermediate key is the *tile_uid* of this record, and the intermediate value is the combination of columns which are specified in the select clause. Note that the intermediate key/values will participate in a two-table join, and a tag must be attached to the value in order to indicate which table the record belongs to. Besides, since the query for this case is a self-join, we use a shared scan in the map function to execute the data filter operations on both instances of the same table. Therefore, a single map input key/value could generate 0, 1 or 2 intermediate key/value pairs, according to the SELECT clause and the values of the record.

The shuffle phase is performed by Hadoop internally, which groups data by tile UIDs. The workflow of the reduce function is shown in the reduce function in Algorithm 2. According to the main structure of the program, the input key of the reduce function is the join key (*tile_uid*), and the input values of the reduce function are all records with the same *tile_uid*. In the reduce function, we first initialize two temporary files, then we dispatch records into corresponding files. After that, we invoke RESQUE engine to build R*-tree indexes

Algorithm 2: MapReduce program for spatial join query

```
function Map(k,v):
    _tile_uid = projectKey(v);
    join_seq = projectJoinSequence(v);
    record = projectRecord(v);
    v = concat(join_seq,record);
    emit(_tile_uid , v);

function JoinReduce(k,v):
    /* arraylist holds join objects      */
    join_set = [ ] ;
    for v_i in v do
        join_seq = projectJoinSequence(v_i);
        record = projectRecord(v_i);
        if join_seq == 0 then
            join_set[0].append(record);
        if join_seq == 1 then
            join_set[1].append(record);
    /* library call to RESQUE            */
    plan = RESQUE.genLocalPlan(join_set);
    result = RESQUE.processQuery(plan);
    for item in result do
        emit(item);
```

and execute the query. The execution result data sets are stored in a temporary in-memory file. Finally we parse that file, and output the result to HDFS. Note that the function *RESQUE.processQuery* here performs multiple spatial opeartions together, including evaluation of WHERE condition, projection, and computation (e.g., *ST_intersection* and *ST_area*), which could be customized.

4.3 Other Spatial Query Types

Other spatial queries can follow a similar process pattern as shown in Algorithm 1. Spatial selection/containment is a simple query type in which objects geometrically contained in selection region are returned. For example, in a medical imaging scenario, users may be interested in the cell features which are contained in a cancerous tissue region. Thus, a user can issue a simple query as shown in Figure 4 to retrieve cancerous cells.

```
1: SELECT * FROM  markup_polygon m, human_markup h
2: WHERE h.name='cancer' AND
3: ST_CONTAINS(h.region, m.polygon) = TRUE;
```

Figure 4: An Example Containment Query in SQL

Since data is partitioned into tiles, containment queries can be processed in a *filter-and-refine* fashion. In the filter step, tiles which are disjoint from the query region can be filtered. In the refinement step, the candidate objects are checked with precise geometry test. The global region index is used to generate a selective table scan operation which only scans the file splits which potentially contain the query results. The query would be translated into a map only MapReduce program shown in Algorithm 3. Support of multi-way spatial join queries and nearest neighbor queries follow a similar pattern and are discussed in our previous work [10]. Figure 4 illustrates a containment query in which the task is to retrieve all the cells which are spatially contained in a region which is marked as cancereous.

Algorithm 3: MapReduce program for containment query

```
function Map(k,v):
    /* a arraylist holds spatial objects */
    candidate_set = [ ];
    _tile_id = projectKey(v);
    for v_i in v do
        record = projectRecord(v_i);
        candidate_set.append(record);
    tile_boundary =getTileBoundary(_tile_id);
    if queryRegion.contains(tile_boundary) then
        emitAll(candidate_set);
    else
        if queryRegion.intersects(tile_boundary) then
            for record in candidate_set do
                if queryRegion.contains(record) then
                    emit(record);
```

Algorithm 4: Boundary aware spatial join processing

```
function Map(k,v):
    _tile_id = projectKey(v);
    record = projectRecord(v);
    if isBoundaryObject(record, _tile_id) then
        tiles = getCrossingTiles(record) ;
        /* replicate to multiple tiles     */
        for tile_id in tiles do
            emit(tile_id , v);

function JoinReduce(k,v):
    /* performs tile based spatial join,
       same as reduce function in
       Algorithm 2                          */

function Map(k,v):
    uid_1 = projectUID(v,1);
    uid_2 = projectUID(v,2);
    key = concat(uid_1,uid_2);
    emit(key,v);
/* Hadoop sorts records by key and
   shuffles them                            */
function Reduce(k,v):
    for records in v do
        if isUniq(record) then
            emit(record);
```

5. BOUNDARY HANDLING

Tile is the basic parallelization unit in Hadoop-GIS. However, in tile based partitioning, some spatial objects may lie on tile boundaries. We define such an object as *boundary object* of which spatial extent crosses multiple tile boundaries. For example, in Figure 5 (left), the object p is a boundary object which crosses the tile boundaries of tiles *S* and *T*. In general, the fraction of boundary objects is inversely proportional to the size of the tile. As tile size gets smaller, the percentage of boundary objects increases.

There are two basic approaches to handle boundary objects. They either have to be specially processed to guarantee query semantics and correctness, or they can be simply discarded. The latter is suitable for a scenario where approximate query results are needed, and query results would not be effected by the tiny fraction of boundary objects. Whereas in many other cases, accurate and consistent query results are required and the boundary objects need to be handled properly.

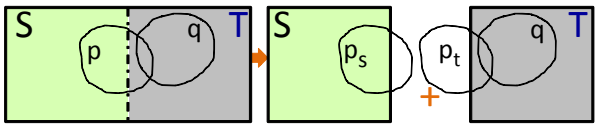


Figure 5: Boundary object illustration

Hadoop-GIS remedies the boundary problem in a simple but effective way. If a query requires to return complete query result, Hadoop-GIS generates a query plan which contains a pre-processing task and a post-processing task. In the pre-processing task, the boundary objects are duplicated and assigned to multiple intersecting tiles (multiple assignment). When each tile is processed independently during query execution, the results are not yet correct due to the duplicates. In the post-processing step, results from multiple tiles will be normalized, e.g., to eliminate duplicate records by checking the *object uid*s, which are assigned internally and globally unique. For example, when processing the spatial join query, the object p is duplicated to tiles *S* and *T* as p_s and p_t (Figure 5 right). Then the same process of join processing follows as if there are no boundary objects. In the post-processing step, objects will go through a filtering process in which duplicate records are eliminated.

Intuitively, such approach would incur extra query processing cost due to the replication and duplicate elimination steps. However, this extra cost is very small compared to the overall query processing time, and we will experimentally quantify such overhead later in Section 7.

6. INTEGRATION WITH HIVE

Hive [29] is an open source MapReduce based query system that provides a declarative query language for users. By providing a virtual table like view of data, SQL like query language HiveQL, and automatic query translation, Hive achieves scalability while it greatly simplifies the effort on developing applications in MapReduce. HiveQL supports a subset of standard ANSI SQL statements which most data analysts and scientists are familiar with.

6.1 Architecture

To provide an integrated query language and unified system on MapReduce, we extend Hive with spatial query support by extending HiveQL with spatial constructs, spatial query translation and execution, with integration of the spatial query engine into Hive query engine (Figure 1). We call the language QL^{SP}, and the Hive integrated version of Hadoop-GIS as $Hive^{SP}$. An example spatial SQL query is shown in Figure 3. The spatial indexing aware query optimization will take advantage of RESQUE for efficient spatial query support in Hive.

There are several core components in $Hive^{SP}$ to provide spatial query processing capabilities. i) *Spatial Query Translator* parses and translates SQL queries into an abstract syntax tree. We extend the HiveQL translator to support a set of spatial query operators, spatial functions, and spatial data types. ii) *Spatial Query Optimizer* takes an operator tree as an input and applies rule based optimizations such as predicate push down or index-only query processing. iii) *Spatial Query Engine* supports following infrastruc-

ture operations: spatial relationship comparison, such as *intersects, touches, overlaps, contains, within, disjoint*, spatial measurements, such as *intersection, union, distance, centroid, area*; and spatial access methods for efficient query processing, such as *R*-Tree*, *Hilbert R-Tree* and *Voronoi Diagram*. The engine is compiled as a shared library and can be easily deployed.

Users interact with the system by submitting SQL queries. The queries are parsed and translated into an operator tree, and the query optimizer applies heuristic rules to the operator tree to generate an optimized query plan. For a query with spatial query operator, MapReduce codes are generated, which call an appropriate spatial query pipeline supported by the spatial query engine. Generated MapReduce codes are submitted to the execution engine to return intermediate results, which can be either returned as final query results or be used as input for next query operator. Spatial data is partitioned based on the attribute defined in the "PARTITION BY" clause and staged to the HDFS.

6.2 Query Processing

Hive^{SP} uses the traditional *plan-first, execute-next* approach for query processing, which consists of three steps: query translation, logical plan generation, and physical plan generation. To process a query expressed in SQL, the system first parses the query and generates an abstract syntax tree. Preliminary query analysis is performed in this step to ensure that the query is syntactically and grammatically correct. Next, the abstract syntax tree is translated into a logical plan which is expressed as an operator tree, and simple query optimization techniques such as predicate push down and column pruning are applied in this step. Then, a physical plan is generated from the operator tree which eventually consists of series of MapReduce tasks. Finally, the generated MapReduce tasks are submitted to the Hive runtime for execution.

Major differences between Hive and Hive^{SP} are in the logical plan generation step. If a query does not contain any spatial operations, the resulting logical query plan is exactly the same as the one generated from Hive. However, if the query contains spatial operations, the logical plan is regenerated with special handling of spatial operators. Specifically, two additional steps are performed to rewrite the query. First, operators involving spatial operations are replaced with internal spatial query engine operators for tile level query processing. Second, serialization/deserialization operations are added before and after the spatial operators to prepare Hive for communicating with the spatial query engine.

An example query plan is given in Figure 6, which is generated from translating SQL query in Figure 3. Notice that the spatial join operator is implemented as reduce side join and the spatial data table is partitioned by tiles for parallel processing, with a user specified partition column during virtual table definition.

6.3 Software

Hadoop-GIS has two forms: the standalone library version which can be invoked through customizations, and Hive integrated version Hive^{SP}. Hive^{SP} is designed to be completely hot-swappable with Hive. Hive users only need to deploy the spatial query engine on the cluster nodes, and turn the spatial query processing switch on. Any query that runs on Hive can run on Hive^{SP} without modification.

The spatial query engine is written in `C++` and compiled as a shared library. We use `libspatial` library [8] for building R*-Tree index, and the library is extended to support index based spatial operations such as two-way, multi-way spatial joins and nearest neighbor search [10].

To use the system, users follow the same user guidelines of using Hive. First of all, users need to create all the necessary table schema which will be persisted to the metastore as metadata. Spatial columns need to be specified with corresponding data types defined in ISO SQL/MM Spatial. Spatial partitioning column can also be specified. After the schema creation, users can load the data through Hive data loading tool. Once data is loaded, users can begin to write spatial SQL queries.

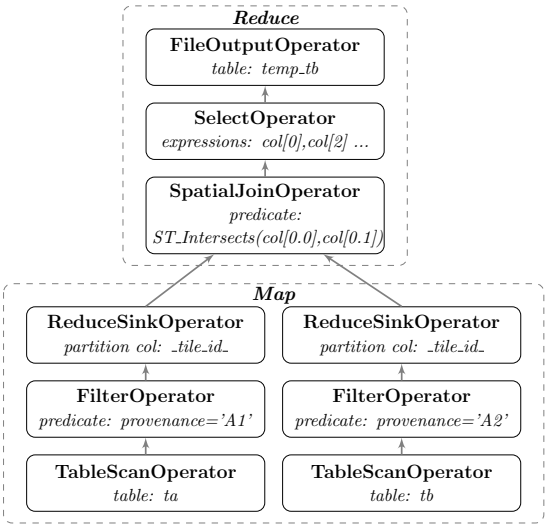


Figure 6: Two-way spatial join query plan

7. PERFORMANCE STUDY

We study the performance of RESQUE query engine versus other SDBMS engines, the performance of Hadoop-GIS versus parallel SDBMS, scalability of Hadoop-GIS in terms of number of reducers and data size, and query performance with boundary handling.

7.1 Experimental Setup

Hadoop-GIS: We use a cluster with 8 nodes and 192 cores. Each of these 8 nodes comes with 24 cores (AMD 6172 at 2.1GHz), 2.7TB hard drive at 7200rpm and 128GB memory. A 1Gb interconnecting network is used for node communication. The OS is CentOS 5.6 (64 bit). We use the Cloudera Hadoop 2.0.0-cdh4.0.0 as our MapReduce platform, and Apache Hive 0.7.1 for Hive^{SP}. Most of the configuration parameters are set to their default value, except the JVM maximum heap size which is set to 1024MB. The system is configured to run a maximum of 24 map or reduce instances on each node. Datasets are uploaded to the HDFS and the replication factor is set to 3 on each datanode.

DBMS-X: To have a comparison between Hadoop-GIS and parallel SDBMS, we installed a commercial DBMS (DBMS-X) with spatial extensions and partitioning capabilities on two nodes. Each node comes with 32 cores, 128GB memory, and 8TB RAID-5 drives at 7200rpm. The OS for the nodes is CentOS 5.6 (64 bit). There are a total of 30 database partitions, 15 logical partitions on each node. With the technical support from the DBMS-X vendor, the parallel SDBMS has been tuned with many optimizations, such as co-location of common joined datasets, replicated spatial reference tables, proper spatial indexing, and query hints. For RESQUE query engine comparison, we also install PostGIS (V1.5.2, single partition) on a cluster node.

7.2 Dataset Description

We use two real world datasets: pathology imaging, and OpenStreetMap.

Pathology Imaging (PI). This dataset comes from image analysis of pathology images, by segmenting boundaries of micro-anatomic objects such as nuclei and tumor regions. The images are provided by Emory University Hospital. Spatial boundaries have been validated, normalized, and represented in WKT format. We have dataset sizes at 1X (18 images, 44GB), 3X (54 images, 132GB), 5X (90 images, 220GB), 10X (180 images, 440GB), and 30X (540 images, 1,320GB) for different testings. The average number of nuclei per image is 0.5 million, and 74 features are derived for each nucleus.

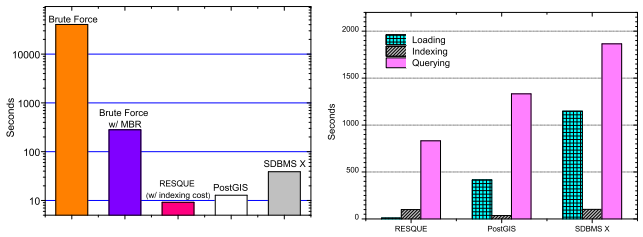
OpenStreetMap (OSM). OSM [6] is a large scale map project through extensive collaborative contribution from a large number of community users. It contains spatial representation of geometric features such as lakes, forests, buildings and roads. Spatial objects are represented by a specific type such as points, lines and polygons. We download the dataset from the official website, and parse it into a spatial database. The table schema is simple and it has roughly 70 columns. We use the polygonal representation table with more than 87 million records. To be able to run our queries, we construct two versions of the OSM dataset, one from a latest version, and another smaller one from an earlier version released in 2010. The dataset is also dumped as text format for Hadoop-GIS.

7.3 Query Benchmarking

We use three typical queries for the benchmark: spatial join (spatial cross-matching), spatial selection (containment query), and aggregation. Many other complex queries can be decomposed into these queries, for example, a spatial aggregation can be run in two steps: first step for spatial object filtering with a containment query, followed by an aggregation on filtered spatial objects. The spatial join query on PI dataset is demonstrated in Figure 3 for joining two datasets with an *intersects* predicate. Another similar spatial join query on OSM dataset is also constructed to find changes in spatial objects between two snapshots. We construct a spatial containment query, illustrated in Figure 4 for PI case, to retrieve all objects within a region, where the containment region covers a large area in the space. A similar containment query is also constructed for OSM dataset with a large query region. For aggregation query, we compute the average area and perimeter of polygons of different categories, with 100 distinct labels.

7.4 Performance of RESQUE Engine

Standalone Performance. An efficient query engine is a critical building block for a large scale system. To test the standalone performance of RESQUE, we run it on a single node as a single thread application. The spatial join query is used as a representative benchmark. We first test the *effect of spatial indexing*, by taking a single tile with two result sets (5506 markups vs 5609 markups), and the results are shown in Figure 8(a). A *brute-force approach* compares all possible pairs of boundaries in a nested loop manner without using any index, and takes 673 minutes. Such slow performance is due to polynomial complexity on pair-wise comparisons and high complexity on geometric intersection testing. An *optimized brute-force approach* will first eliminate all the non-intersecting markup pairs by using a MBR based filtering. Then it applies the geometry intersection testing algorithm on the candidate markup pairs. This approach takes 4 minutes 41 seconds, a big saving with minimized geometric computations. Using RESQUE with indexing based spatial join, the number of computations is significantly reduced, and it only takes 10 seconds. When profiling the cost for RESQUE, we observe that reading and parsing cost is 30%, R*-Tree construction cost is 0.2%, MBR filtering cost is 0.67%, and spatial refinement and measurement cost is 69.13%. With fast development of CPU speed, spatial index construction takes very little time during the query process, which motivates us to develop an index-on-demand approach to support spatial queries. We can also see that geometric computation dominates the cost, which can be accelerated through parallel computation on a cluster.



(a) Single tile (b) Single image

Figure 8: Performance of RESQUE

Data Loading Efficiency. Another advantage of RESQUE is its light data loading cost compared to SDBMS approach. We run three steps to get the overall response time (data loading, indexing and querying) on three systems: RESQUE on a single slot MapReduce with HDFS, PostGIS and SDBMS-X with a single partition. The data used for the testing are two results from a single image (106 tiles, 528,058 and 551,920 markups respectively). As shown in Figure 8(b), data loading time for REQUE is minimal compared to others. With controlled optimization, RESQUE outperforms on the overall efficiency.

7.5 Performance of Hadoop-GIS

7.5.1 *Hadoop-GIS versus Parallel SDBMS*

For the purpose of comparison, we run the benchmark queries on both Hadoop-GIS and DBMS-X on the PI dataset. The data is partitioned based on tile UIDs – boundary objects are ignored in the testing as handling boundary objects in SDBMS is not supported directly. Figure 7 shows the performance results. The horizontal axis represents the number of parallel processing units (PPU), and the vertical axis represents query execution time. For the parallel SDBMS, the number of PPUs corresponds to the number of database partitions. For Hadoop-GIS, the number of PPU corresponds to the number of mapper and reducer tasks.

Join Query. As the figure 7(a) shows, both systems exhibit good scalability, but overall Hadoop-GIS performs much better compared to DBMS-X, which has already been well tuned by the vendor. Across different numbers of PPUs, Hadoop-GIS is more than a factor of two faster than DBMS-X. Given that DBMS can intelligently place the data in storage and can reduce IO overhead by using index based record fetching, it is expected to have better performance on IO heavy tasks. However, a spatial join involves expensive geometric computation, and the query plan generated by the DBMS is suboptimal for such tasks. Another reason for the performance of DBMS-X is because of its limited capability on handling computational skew, even though the built-in partitioning function generates a reasonably balanced data distribution. Hadoop has an on-demand task scheduling mechanism which can help alleviate such computational skew.

Containment Query. For containment queries, Hadoop-GIS outperforms DBMS-X on a smaller scale and has a relatively flat performance across different number of PPUs. However, DBMS-X

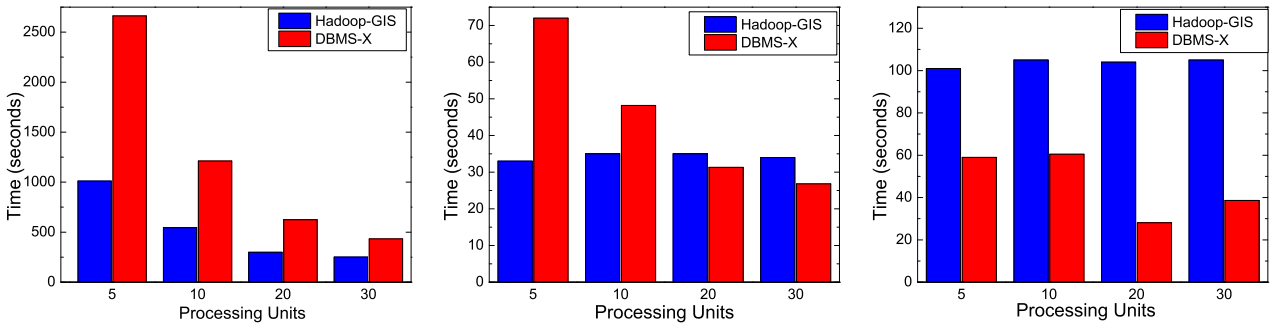


Figure 7: Performance Comparison between Hadoop-GIS and DBMS-X on PI Dataset

exhibits better scalability when scaled out with larger number of partitions. Recall that, in Hadoop-GIS, a containment query is implemented as a Map only MapReduce job, and the query itself is less computationally intensive compared to the join query. Therefore, the time is actually being spent on reading in a file split, parsing the objects, and checking if the object is contained in the query region. On the other hand, DBMS-X can take advantage of a spatial index and can quickly filter out irrelevant records. Therefore it is not surprising that DBMS-X has slightly better performance for containment queries.

Aggregation Query. Figure 7(c) demonstrates that DBMS-X performs better than Hadoop-GIS on aggregation task. One reason for this is that Hadoop-GIS has the record parsing overhead. Both systems have similar query plans - a whole table scan followed by an aggregation operation on the spatial column, which have similar I/O overhead. In Hadoop-GIS, however, the records need to be parsed in real-time, whereas in DBMS-X records are pre-parsed and stored in binary format.

In a summary, Hadoop-GIS performs better in compute-intensive analytical tasks and exhibits nice scalability - a highly desirable feature for data warehousing applications. Moreover, it needs much less tuning effort compared to the database approach. However, MapReduce based approach may not be the best choice if the query task is small, e.g., queries to retrieve a small number of objects.

7.5.2 *Performance on OpenStreetMap Dataset*

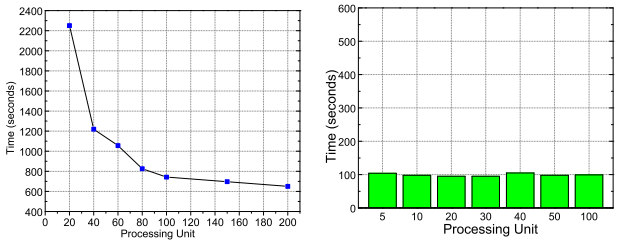


Figure 9: Performance of Hadoop-GIS on OSM

We have also tested performance and scalability of Hadoop-GIS on OSM dataset, which contains geospatial objects. To test system scalability, we run the same types of queries as on the PI dataset. For the join query, the query returns objects which have been changed in the newer version of the dataset. Therefore, the join predicate becomes *ST_EQUAL = FALSE*.

Figure 9 shows the performance results for Hadoop-GIS on OSM dataset. As Figure 9(a) shows, Hadoop-GIS exhibits very nice scalability on the join task. When the number of available processing units is increased to 40 from 20, the query time nearly reduced into half, which is almost a linear speed-up. With increase of the number of PPUs, there is a continuous drop on the query time.

Figure 9(b) illustrates the containment query performance on OSM dataset, where running time is less than 100 seconds. The variance of query performance across different number of PPUs is flat, due to the fact that containment queries are relatively IO intensive.

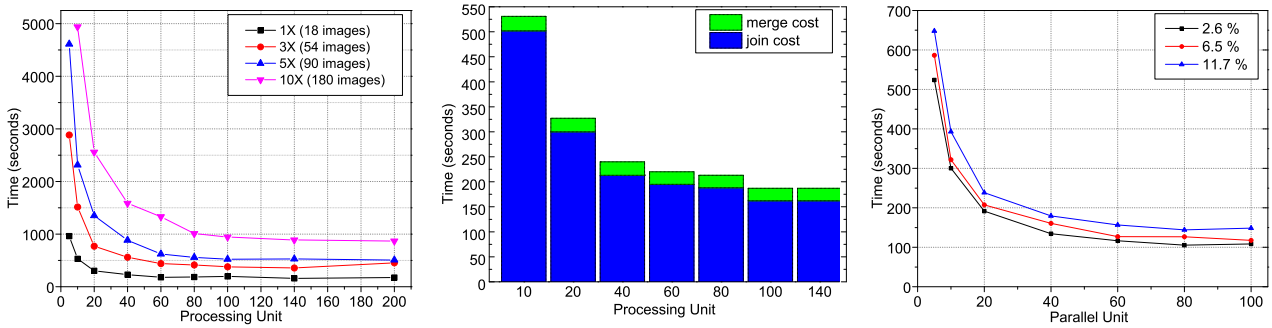
7.5.3 *Scalability of Hadoop-GIS*

Figure 10(a) shows the scalability of the system. Datasets used include: 1X, 3X, 5X, and 10X datasets, with varying number of PPUs. We can see a continuous decline of query time when the number of reducers increases. It achieves a nearly linear speed-up, e.g., time is reduced to by half when the number of reducers is increased from 20 to 40. The average querying time per image is about 9 seconds for the 1X dataset with all cores, comparing with 22 minutes 12 seconds in a single partition PostGIS. The system has a very good scale up feature. As the figure shows, query processing time increases linearly with dataset size. The time for processing the join query on 10X dataset is roughly 10 times of the time for processing a 1X dataset.

7.6 Boundary Handling Overhead

We run the join query on PI dataset to measure the overhead in boundary handling step. Figure 10(b) shows the performance of spatial join query with boundary handling. The blue bars represent the cost of processing the query, and the green bars represent the cost of amending the results. As the figure shows, the cost of boundary handling is very small. Boundary handling overhead depends on two factors – the number of boundary objects and the size of the query output. If the number of objects on the tile boundary accounts for a considerable fraction of the dataset, the overhead should not dominate the query processing time. Therefore, we test the join query on the same dataset in which the number of boundary objects is deliberately increased. Figure 10(c) shows the spatial join query performance with different fraction of boundary objects. The lines represent query performance with varying percentage of boundary objects as shown in the legend. It is clear from the figure that, the boundary handling overheard increases linearly with the percentage of boundary objects.

In Figure 10(c), we show the performance when the percentage of boundary objects is as high as 11.7%. In reality, the number of objects is much less than this number. We have performed an experiment with the OSM dataset in which the dataset is partitioned into 1 million tiles. Even in such a fine granular level of partitioning, the number of objects lying on the grid boundary is less than 2%.



(a) Scalability Test (b) Boundary handling overhead (c) Overhead variation

Figure 10: Scalability of Hadoop-GIS & boundary handling overhead

8. RELATED WORK

Parallel SDBMS has been used for managing and querying large scale spatial data based on shared nothing architecture [**?**], such as Greenplum, IBM Netezza, Teradata, and partitioned version of IBM DB2 Spatial Extender, Oracle Spatial, MS SQL Server Spatial, and PostGIS. These approaches lack the framework for spatial partitioning and boundary object handling. Data loading speed is a major bottleneck for SDBMS based solutions [26], especially for complex structured spatial data types [31]. We have previously developed a parallel SDBMS based approach *PAIS* [30, 31, 7] based on DB2 DPF with reasonable scalability, but the approach is highly expensive on software license and hardware requirement[26], and requires sophisticated tuning and maintenance. The objective of the work presented in this paper is to provide a scalable and cost effective approach to support expressive and high performance spatial queries. The Sloan Digital Sky Survey project (SDSS) [3] creates a high resolution multi-wavelength map of the Northern Sky with 2.5 trillion pixels of imaging, and takes a large scale parallel database approach. SDSS provides a high precision GIS system for astronomy, implemented as a set of UDFs. The database runs on GrayWulf architecture [28] through collaboration with Microsoft.

Partitioning based approach for parallelizing spatial joins is discussed in [36], which uses the multiple assignment, single join approach with partitioning-based spatial join algorithm. The authors also provide re-balancing of tasks to achieve better parallelization. We take the same multiple assignment approach for partitioning, but use index based spatial join algorithm, and rely on MapReduce for load balancing. R-Tree based parallel spatial join is also proposed in early work [15] with a combined shared virtual memory and shared nothing architecture. Recently we have exploited massive data parallelism by developing GPU aware parallel geometric computation algorithms to support spatial joins running on desktop machines [32]. Integrating GPU into our MapReduce pipeline is among our future work.

Spatial support has been extended to NoSQL based solutions, such as neo4j/spatial [2] and GeoCouch [1]. These approaches build spatial data structures and access methods on top of key-value stores, thus take advantage of the scalability. However, these approaches support limited queries, for example, GeoCouch supports only bounding box queries, and there is no support of the analytical spatial queries for spatial data warehousing applications.

In [16], an approach is proposed on bulk-construction of R-Trees through MapReduce. In [34], a spatial join method on MapReduce is proposed for skewed spatial data, using an in-memory based strip plane sweeping algorithm. It uses a duplication avoidance technique which could be difficult to generalize for different query types. Hadoop-GIS takes a hybrid approach on combining partitioning with indexes and generalizes the approach to support multiple query types. Besides, our approach is not limited to memory size. VegaGiStore [35] tries to provide a Quadtree based global partitioning and indexing, and a spatial object placement structures through Hibert-ordering with local index header and real data. The global index links to HDFS blocks where the structures are stored. It is not clear how boundary objects are handled in partitioning, and how parallel spatial join algorithm is implemented. Work in [4] takes a fixed grid partitioning based approach and uses sweep line algorithm for processing distributed joins on MapReduce. It is unclear how the boundary objects are handled, and no performance study is available at the time of evaluation. The work in [20] presents an approach for multi-way spatial join for rectangle based objects, with a focus on minimizing communication cost. A MapReduce based Voronoi diagram generation algorithm is presented in [11]. In our work in [10], we present preliminary results on supporting multi-way spatial join queries and nearest neighbor queries for pathology image based applications. This paper has significantly generalized the work to provide a generic framework for supporting multiple types of spatial applications, and a systematic approach for data partitioning and boundary handling.

Comparisons of MapReduce and parallel databases for structured data are discussed in [26, 18, 27]. Tight integration of DBMS and MapReduce is discussed in [9, 33]. MapReduce systems with high-level declarative languages include Pig Latin/Pig [25, 19], SCOPE [17], and HiveQL/Hive [29]. YSmart provides an optimized SQL to MapReduce job translation and is recently patched to Hive. Hadoop-GIS takes an approach that integrates DBMS's spatial indexing and declarative query language capabilities into MapReduce.

9. CONCLUSION AND DISCUSSION

"Big" spatial data from imaging and spatial applications share many similar requirements for high performance and scalability with enterprise data, but has its own unique characteristics – spatial data are multi-dimensional and spatial query processing comes with high computational complexity. In this paper, we present Hadoop-GIS, a solution that combines the benefit of scalable and cost-effective data processing with MapReduce, and the benefit of efficient spatial query processing with spatial access methods. Hadoop-GIS achieves the goal through spatial partitioning, partition based parallel processing over MapReduce, effective handling of boundary objects to generate correct query results, and multi-level spatial indexing supported customizable spatial query engine. Our experiment results on two real world use cases demonstrate that Hadoop-GIS provides a scalable and effective solution for analytical spatial queries over large scale spatial datasets.

Our work was initially motivated by the use case of pathology

imaging. We started from a parallel SDBMS based solution [31] and experienced major problems such as the data loading bottleneck, limited support of complex spatial queries, and the high cost of software and hardware. Through the development and deployment of MapReduce based query processing, we are able to provide scalable query support with cost-effective architecture [10]. In this paper, we have generalized the approach to provide a solution that can be used to support various spatial applications.

Ongoing work includes support of 3D pathology analytical imaging. 3D examinations of tissues at microscopic resolution are now possible and have significant potential to enhance the study of both normal and disease processes, by exploring structural changes or spatial relationship of disease features. A single 3D image consists of around a thousand slices and consumes more than 1TB storage, and spatial queries on massive 3D geometries pose many new challenges. Another ongoing work is to support queries for global spatial patterns such as density and directional patterns. We are also working on integrating GPU based spatial operations [32] into the query pipeline.

10. ACKNOWLEDGMENTS

This work is supported in part by PHS Grant UL1RR025008 from the CTSA program, R24HL085343 from the NHLBI, by Grant Number R01LM009239 from the NLM, by NCI Contract No. N01-CO-12400 and 94995NBS23 and HHSN261200800001E, by NSF CNS 0615155, CNS-1162165, 79077CBS10, CNS-0403342, OCI-1147522, and P20 EB000591 by the BISTI program. IBM provides academic license for DB2. David Adler and Susan Malaika from IBM provided many insightful suggestions.

11. REFERENCES

[1] Geocouch. `https://github.com/couchbase/geocouch`.
[2] neo4j/spatial. `https://github.com/neo4j/spatial`.
[3] The sloan digital sky survey project (sdss). `http://www.sdss.org`.
[4] Spatialhadoop. http://spatialhadoop.cs.umn.edu/.
[5] Hadoop-gis wiki. `https://web.cci.emory.edu/confluence/display/hadoopgis`, 2013.
[6] Openstreetmap. `http://www.openstreetmap.org`, 2013.
[7] Pathology analytical imaging standards. https://web.cci.emory.edu/confluence/display/PAIS, 2013.
[8] libspatialindex. `http://libspatialindex.github.com`.
[9] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, Aug. 2009.
[10] A. Aji, F. Wang, and J. H. Saltz. Towards Building A High Performance Spatial Query System for Large Scale Medical Imaging Data. In *SIGSPATIAL/GIS*, pages 309–318. ACM, 2012.
[11] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based geospatial query processing with mapreduce. In *CLOUDCOM*, pages 9–16, 2010.
[12] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.
[13] J. V. d. Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *VLDB*, pages 461–470, 2001.
[14] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, 2010.
[15] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using r-trees. In *ICDE*, 1996.
[16] A. Cary, Z. Sun, V. Hristidis, and N. Rishe. Experiences on processing spatial data with mapreduce. In *SSDBM*, pages 302–319, 2009.
[17] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.
[18] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.
[19] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high level dataflow system on top of MapReduce: The Pig experience. *PVLDB*, 2(2):1414–1425, 2009.
[20] H. Gupta, B. Chawda, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. Mohania. Processing multi-way spatial joins on map-reduce. In *EDBT*, pages 113–124, 2013.
[21] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *VLDB*, pages 500–509, 1994.
[22] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, 2011.
[23] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *SIGMOD*, pages 247–258, 1996.
[24] M. A. Nieto-Santisteban, A. R. Thakar, and A. S. Szalay. Cross-matching very large datasets. In *NSTC NASA Conference*, 2007.
[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.
[26] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.
[27] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.
[28] A. S. Szalay, G. Bell, J. vandenBerg, A. Wonders, R. C. Burns, D. Fay, J. Heasley, T. Hey, M. A. Nieto-Santisteban, A. Thakar, C. v. Ingen, and R. Wilton. Graywulf: Scalable clustered architecture for data intensive computing. In *HICSS*, pages 1–10, 2009.
[29] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.
[30] F. Wang, J. Kong, L. Cooper, T. Pan, K. Tahsin, W. Chen, A. Sharma, C. Niedermayr, T. W. Oh, D. Brat, A. B. Farris, D. Foran, and J. Saltz. A data model and database for high-resolution pathology analytical image informatics. *J Pathol Inform*, 2(1):32, 2011.
[31] F. Wang, J. Kong, J. Gao, D. Adler, L. Cooper, C. Vergara-Niedermayr, Z. Zhou, B. Katigbak, T. Kurc, D. Brat, and J. Saltz. A high-performance spatial database based approach for pathology imaging algorithm evaluation. *Journal of Pathology Informatics*, 4(5), 2013.
[32] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. *Proc. VLDB Endow.*, 5(11):1543–1554, 2012.
[33] Y. Xu, P. Kostamaa, and L. Gao. Integrating hadoop and parallel dbms. In *SIGMOD*, pages 969–974, 2010.
[34] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, 2009.
[35] Y. Zhong, J. Han, T. Zhang, Z. Li, J. Fang, and G. Chen. Towards parallel spatial query processing for big spatial data. In *IPDPSW*, pages 2085–2094, 2012.
[36] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 2:175–204, June 1998.