A parallel spatial data analysis infrastructure for the Cloud

Suprio Ray, Bogdan Simion, Angela Demke Brown, Ryan Johnson
Department of Computer Science, University of Toronto
{suprio, bogdan, demke, ryan.johnson} @cs.toronto.edu

ABSTRACT

Spatial data analysis applications are emerging from a wide range of domains such as building information management, environmental assessments and medical imaging. Time-consuming computational geometry algorithms make these applications slow, even for medium-sized datasets. At the same time, there is a rapid expansion in available processing cores, through multicore machines and Cloud computing. The confluence of these trends demands effective parallelization of spatial query processing. Unfortunately, traditional parallel spatial databases are ill-equipped to deal with the performance heterogeneity that is common in the Cloud.

We introduce Niharika, a parallel spatial data analysis infrastructure that exploits all available cores in a heterogeneous cluster. Niharika first uses a declustering technique that creates balanced spatial partitions. Then, Niharika adapts to performance heterogeneity and processing skew in the spatial dataset using dynamic load-balancing. We evaluate Niharika with three load-balancing algorithms and two different spatial datasets (both from TIGER) using Amazon EC2 instances. Niharika adapts to the performance heterogeneity in the EC2 nodes, thereby achieving excellent speedups (e.g., 63.6X using 64 cores on 16 4-core EC2 nodes, in the best case) and outperforming an approach that does not adapt.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Performance, Measurement, Experimentation, Algorithms

Keywords

Spatial join, Cloud, performance heterogeneity, load balancing

1. INTRODUCTION

Spatial analysis applications are rapidly gaining in importance, fueled by the explosive growth in vector spatial data and the availability of spatial features in commercial databases. Alongside traditional uses such as land surveys, city planning and environmental risk assessment, new classes of spatial applications are emerging in domains as diverse as building information management and medical image analysis. These applications perform complex analyses of spatial datasets to provide vital information to governments and industries ranging from insurance to natural resource development to medical diagnostics. In contrast to geospatial Web services such as Google Maps, which are driven by short-running range queries, these spatial analytics applications are characterized by long-running compute-intensive spatial join queries. For example, a spatial join of a polyline table (73 million records representing the contiguous USA) with itself takes roughly *20 hours* to complete on an Amazon EC2 m1.xlarge instance.

The current surge in available processing cores, through multicore architectures and the Cloud computing model, presents an opportunity to dramatically reduce the latency of spatial join queries by creating a scalable parallel data processing infrastructure. The key challenge is how to balance the spatial processing load across a large number of worker nodes, given significant performance heterogeneity in the nodes and processing skew in the workload.

Modern data-centers have evolved as very large distributed systems built from hundreds of machines from multiple hardware generations. Performance heterogeneity among these machines occurs naturally, as disks or nodes fail and are replaced with newer components. The same effect occurs in smaller clusters within an organization. Cloud computing adds a new dimension, in that the infrastructure itself is no longer fixed. Cloud infrastructure providers, such as Amazon EC2, offer different instance types, each with different processing and storage capacity. Furthermore, due to virtualization overhead and load consolidation, different machines of a single instance type may vary widely in performance.

Although the issue of performance heterogeneity has received widespread attention in distributed data processing systems such as MapReduce, not much work has been done in parallel databases in this context. The challenges with parallel query processing in a heterogeneous cluster were recognized by Mayr et al. [14], but their evaluation was limited to User Defined Functions (UDFs) for traditional workloads in a 2 node system. Spatial database workloads are more compute-intensive than traditional database workloads [20] and the amount of computation required to evaluate a spatial join predicate can vary widely for different tuples. Thus, assigning the same number of tuples to each node, even when the nodes are identical, does not guarantee good load balancing. Due to this processing skew, the performance heterogeneity may get aggravated. Moreover, traditional database systems are unable to fully exploit multicore machines [11]. Most databases, including PostgreSQL, do not yet support intra-query parallelism. Although

SIGSPATIAL '13, November 05-08, 2013, Orlando, FL, USA
 ACM 978-1-4503-2521-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2525314.2525347

a few recent projects [2, 4] explored support for intra-query parallelism, none of them looked at spatial join queries.

Our solution is *Niharika*, a distributed spatial query processing system that provides a framework for spatial declustering and dynamic load balancing on top of a cluster of worker nodes, each of which runs a standard PostgreSQL/PostGIS relational database. The overall architecture of Niharika is inspired by HadoopDB [1], which aims to provide a parallel database implementation for Cloud computing. However, Niharika uses a multi-round query execution model that is better suited to exploit multiple processing cores and to address performance heterogeneity. Niharika's mechanisms for data partitioning and dynamic load balancing directly support intra-query parallelism, by concurrently executing multiple customized queries on multiple cores in each machine. A key advantage of our approach is that it requires no change in the underlying relational database engine. By taking advantage of the optimized relational database systems Niharika offers good query execution performance and scalability. Our contributions are as follows:

1. We show how our system is able to exploit multiple processing cores in each node while executing long-running spatial join queries. Niharika is able to achieve near linear speedup (in the number of cores) for most queries and scales to the number of cores in a cluster of multi-core machines.
2. We introduce novel dynamic load-balancing techniques that allocate well-balanced workload partitions to nodes, based on node capability and dynamic load characteristics rather than static partitioning, which enables Niharika to adapt to machine performance heterogeneity We perform extensive performance evaluation experiments with Amazon EC2 M1 Extra Large instances.
3. With a number of experiments we demonstrate the scalability of Niharika on two datasets, one that fits in memory and another that does not fit in memory.

We begin with the motivation for parallelizing spatial join in Section 1.1, and the challenges of heterogeneity and processing skew in Section 1.2, before presenting the Niharika architecture and our spatial declustering approach in Section 2. Section 3 details Niharika's scheduling algorithms for load balancing and data placement, which are evaluated experimentally in Section 4. Related work and conclusions are in Sections 5 and 6, respectively.

1.1 A case for parallelizing spatial join queries

Spatial join queries are used to combine two different datasets based on a spatial predicate. For instance, a polygon dataset representing land use can be joined with another polygon dataset of flood-plains to determine flood-risk areas [19]. To find river bridges, a polyline dataset of roads can be joined with another polyline dataset representing hydrography. Table 1 shows some use cases for spatial join queries. These queries involve computational geometry algorithms to evaluate the relationships between spatial data types. These geometric computations on datasets with many records impose a high computational load, even with spatial indexes, leading to very long query latencies.

To illustrate the performance of the spatial join queries, we selected seven representative medium and long running queries from the Jackpine spatial benchmark's micro-benchmark suite [19]. These queries perform spatial join operations using different topological relations on the edges (polylines) and area (landmass and water polygons) tables from our datasets. We use PostgreSQL with the PostGIS spatial extension as the relational database. We use two real-world spatial datasets that contain diverse geographical features, drawn from the TIGER® data [21], produced by the United States (US) Census Bureau. This is a public domain data source available for each US state. The first dataset consists of the polyline, polygon and point shapefiles for all the counties of California. The second is a much larger dataset covering the contiguous US. Full details of the experimental setup and dataset are in Section 4.1.

[1]We use Line to refer to polylines and Area to refer to polygons.

Table 1: Some use cases of spatial join queries

Use case	Queries
Flood Risk Analysis	Line/Area Intersects Line/Area[1]
	Area Overlaps Area
Land Info Management	Area Overlaps Area
	Area Intersects Area
Medical Imaging	Area Overlaps Area
	Area Intersects Area
Building Info Management	Line/Area Intersects Line/Area
	Line/Area Touches Line/Area
Water/Gas Utilities	Line Crosses Line
	Line Touches Line

Table 2: Jackpine queries and abbreviations with two datasets

Description	California	US
Line Intersects Area (edges and arealm)	LiAca	LiAus
Line Touches Area (edges and arealm)	LtAca	LtAus
Line Crosses Area (edges and arealm)	LcAca	LcAus
Line Intersects Line (edges and edges)	LiLca	LiLus
Line Crosses Line (edges and edges)	LcLca	LcLus
Area Overlaps Area (areawater and areawater)	AoAca	AoAus
Area Touches Area (areawater and areawater)	AtAca	AtAus

Table 2 summarizes these queries on the two datasets, and the abbreviations that we use to refer to them in the text. Note that in the original Jackpine benchmark, the "Line and Line" queries were limited to return 5 result records due to very long execution times; we remove this limit in our experiments.

To establish a baseline for our work, we evaluated the single-node PostGIS performance for the queries in Table 2 on 16 m1.xlarge instances on Amazon EC2. We ran Ubuntu 10.04 Lucid 64-bit with kernel version 2.6.32-33-generic as the OS on each machine. The best (minimum) and worst (maximum) observed execution times are shown in Figure 1. As can be seen some of the queries are very long running. For instance, the AoAus query takes 2 hours (7341 seconds), whereas the LcLus takes over 20 hours (71159 seconds) on average. Thus, it is natural to try to parallelize the spatial join queries, in light of the abundance of processing cores per machine.

Figure 1 also shows that there is a significant difference between the best and worst observed query execution times. This disparity is indicative of performance heterogeneity, which we discuss next.

1.2 Heterogeneity in computing clusters

Both modern data centers and smaller local clusters are typically built from commodity machines. Over time, new machines are added to deal with increasing loads or to replace failed nodes. These new nodes will usually come from newer hardware generations and will have many differences including different CPU models, numbers of cores, cache sizes, and disk models. Even without adding new nodes to the cluster, failed disks may be replaced with newer models, delivering higher performance. Such disk upgrades or replacements are commonplace in real-world clusters.

Performance heterogeneity in a homogeneous cluster can also arise because system performance may degrade over time. Disk performance degradation due to partial media failure (i.e., bad sectors) is a common phenomenon. Therefore, to maintain performance homogeneity over time, a full replacement of the system would be required, which is prohibitively expensive.

Recently, Borthakur [6] observed that, in Facebook's datacen-

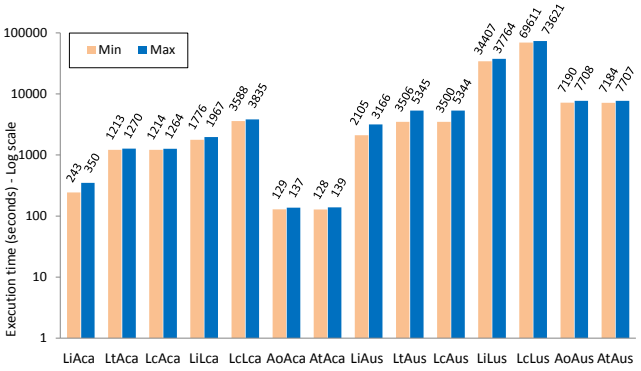


Figure 1: Min and max single-node PostGIS execution times for Jackpine queries, observed on 10 distinct EC2 m1.xlarge instances (warm runs)

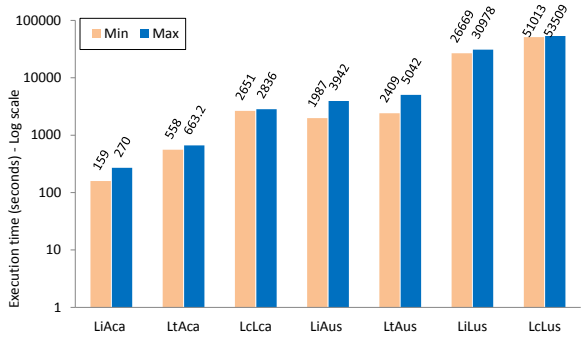


Figure 2: Min and max observed execution times of 7 Jackpine queries in a cluster of local nodes (cold runs)

ters, heterogeneity is the norm and anomalous behavior is far more common than complete failure; at any given time, perhaps 10% of machines in the cluster run 50% slower than the others. This observation suggests that performance heterogeneity in large clusters of machines is more severe than academic researchers usually assume.

In Cloud environments, additional factors such as virtualization overheads, contention for shared physical resources from multiple virtual machine (VM) instances, and the effects of VM migration, can also contribute to performance heterogeneity. Figure 1 shows significant differences between the fastest and slowest execution times, even though we used warm runs and the instances had enough RAM to hold the California dataset (the US dataset also benefited from OS caching). For instance, with the "Line and Area" queries on the US dataset, the slowest node takes roughly 50% longer than the fastest node (1844s extra for LcAus). We also found that the slowest instance varied across different queries, which indicates that the degree of performance heterogeneity may vary dynamically in Cloud environments, as others have observed [10].

Perhaps more surprisingly, we also observed performance heterogeneity in an apparently homogeneous local cluster. We show the min and max execution times of 7 Jackpine queries from Table 2 on 4 distinct local nodes. Each local machine includes 8 Intel Xeon CPU cores (model E8400), 6 MB cache, 2 GB memory and an 880 GB 7200-RPM SATA disk. Every query was run in isolation on an instance of PostgreSQL running on each of these machines. For these experiments, we use cold runs to include disk effects.

As can be seen, there is a significant difference between the best and worst times for the queries. For instance, the LiAca (Line Intersects Area, California dataset) query took 111 seconds longer on the slowest node than on the fastest node. For LtAus (Line Touches Area, USA), the difference was 2633 seconds. On investigation, we found that the cluster was not truly homogeneous: the disk models differ due to replacements, which are common in real-world clusters. In this case, the node with the highest disk read bandwidth was always the first to complete the query.

We now consider the effect of processing skew within the spatial dataset itself.

1.3 Processing skew and load balancing

In a parallel query execution system operating on traditional workloads, the dataset is statically distributed to nodes using either range partitioning or hash partitioning, so that each node can work on a disjoint portion of the dataset. However, these partitioning schemes are not suitable for spatial data because, to execute a spatial join involving two tables, the tables need to be partitioned on the same spatial boundaries. The process of partitioning spatial datasets, known as spatial declustering, involves dividing the spatial domain into two-dimensional disjoint subspaces and assigning them to individual shared-nothing database nodes so that the load is evenly distributed. However, due to the nature of spatial datasets, spatial declustering approaches suffer from a few issues caused by skew. The first issue is the variation in the number of records among different partitions, known as tuple distribution skew, The second issue is the processing skew caused by variation in the size of objects.

Round-robin [16] is a popular spatial declustering technique. It creates a large number of disjoint partitions and then maps them to the nodes in a round robin fashion. Intuitively, this scheme reduces tuple distribution skew, since neighboring partitions which are likely to have similar densities of objects are assigned to different nodes. However, the processing skew cannot be avoided without refactoring the data objects.

To illustrate the problem, we calculated a frequency histogram of the total area of all the polygons in the partitions of the arealm_ca table. In over 96% of the partitions, we found that the total area of all polygons is less than 0.25 million hectares. However, for three partitions the total area is over 1.5 million hectares, because they contain very large objects such as the Death Valley National Park. Moreover, several large polygons usually cluster within the same spatial partition. These large objects require expensive refinement processing more frequently, since their minimum bounding rectangle (MBR) interacts with a larger number of other objects. The TIGER dataset already does a good job of refactoring very long line features (such as rivers, or roads) using polylines, consisting of many smaller line segments. Thus, the processing skew is less pronounced in the "Line and Line" queries.

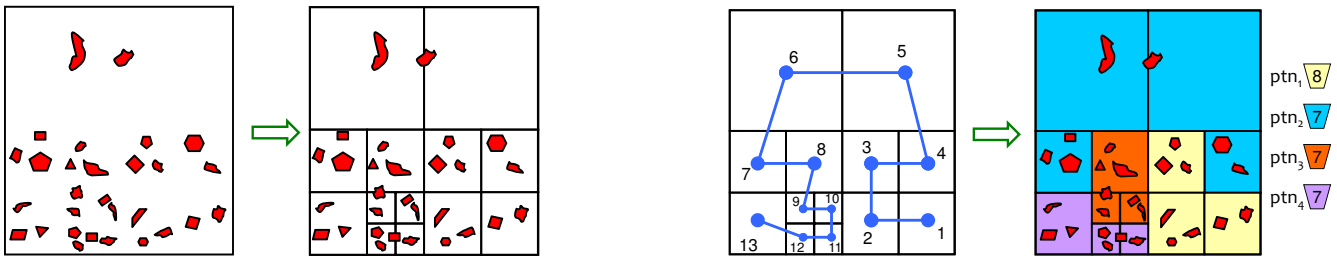
In a real-world cluster, the combined effects of processing skew and machine performance heterogeneity make load balancing even more challenging. Therefore, significant opportunity exists to improve load balancing, which would in turn reduce query execution time and achieve better speedup.

2. OVERVIEW OF NIHARIKA

Niharika is a parallel spatial query execution infrastructure that aims to accelerate the execution of long running spatial analysis queries by exploiting all available processing cores. It addresses the challenges of performance heterogeneity and processing skew by combining a data partitioning scheme and task scheduling with dynamic load balancing. This mechanism can naturally take advantage of the multiple processing cores in each machine.

2.1 Architecture

Niharika uses a master-slave architecture, inspired by HadoopDB, with a task scheduler called the Coordinator, and worker nodes called DBWorkers, as shown in Figure 3. The Coordinator handles



(a) Recursive tiling (b) Declustering

Figure 4: Spatial partitioning (a) and declustering using Hilbert SFC traversal and tile aggregation into partitions (b)

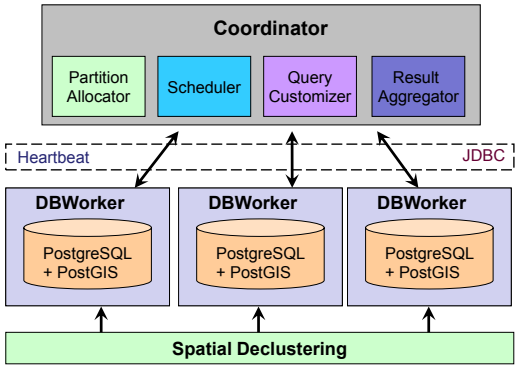


Figure 3: Architecture of Niharika

the scheduling of query jobs, while DBWorkers run tasks locally. Each DBWorker node hosts a PostgreSQL/PostGIS instance, which is used to locally process its portion of the dataset.

The role of the Coordinator resembles that of the JobTracker in HadoopDB. It pushes the query execution into the database instances on each DBWorker, thus leveraging the highly-optimized RDBMS processing. The Coordinator performs aggregation and group-by operation on the resultset returned by the DBWorkers. The coordinator also eliminates duplicates from the resultset.

We implement a spatial declustering scheme (see Section 2.2), which aims to assign neighboring tiles to the same node as much as possible, while still reducing tuple distribution skew.

The *Scheduler* component of the Coordinator is responsible for making these assignments, with the help of a *Query Customizer* that re-writes the submitted query for execution on each node so that nodes only process records from their assigned partitions. We implement and evaluate three algorithms for dynamic load balancing within this framework.

2.2 Spatial declustering in Niharika

We wanted to use a declustering approach that distributes the records as evenly as possible among the resultant spatial partitions. The number of partitions and the order of the distribution (which affects locality) are important considerations. If we have a small number of large partitions, tuple distribution skew is more likely, leading to load imbalance. On the other hand, a large number of partitions increases duplication of records because each spatial object (line or polygon) that is not completely contained within a single partition must be replicated to all partitions that it overlaps.

Niharika's spatial declustering algorithm attempts to equalize the number of tuples assigned to each node, instead of mapping an equal number of partitions to each node, thereby minimizing tuple distribution skew. It also reduces duplicates by taking advantage of the locality-preserving property of Space-filling curves (SFC) when distributing partitions to nodes. Our algorithm has three phases:

Phase 1 - recursive tiling: The spatial domain is recursively subdivided into small disjoint subspaces, called tiles, so that the number of spatial items contained in each tile is expected to be roughly equal. In each step, if the number of spatial objects that are mapped to each tile exceeds a threshold, the tile is split into 4 equal parts. This is essentially a quad-tree splitting scheme, illustrated in Figure 4(a). The end result is that each tile contains roughly the same number of objects. The threshold on the maximum number of objects per tiles is chosen to be a relatively small value, so that many tiles are generated. This decreases tuple distribution skew.

Phase 2 - SFC order: After the tiles are generated, a Hilbert SFC traversal order is imposed on them (see Figure 4(b), left-side), to preserve locality when assigning tiles to nodes. The Hilbert curve can achieve near-optimal locality, however, it is often not used due to the complexity of the traversal operations. We use a fast look-up scheme [8] that is well-suited for large datasets.

Phase 3 - tile aggregation: Phase 1 and 2 provide a partitioning scheme that ensures (1) that the number of objects in each tile will be less than a threshold and (2) good spatial locality of the objects. But, in some tiles the number of objects can be close to the maximum threshold and in others it could be close to zero. Since the tiles are mapped to the nodes with Hilbert SFC, some nodes could end up with tiles corresponding to relatively feature-sparse geographic regions, resulting in significant tuple distribution skew.

To further minimize this skew, we perform tile aggregation. Instead of distributing tiles directly to the nodes, we accumulate tiles in partitions until nearly filled and then distribute the partitions to the nodes. Our algorithm resembles the next fit online bin packing problem, since the tiles must be distributed by maintaining a particular SFC traversal order. However, we constrain the maximum number of available partitions. The algorithm runs iteratively until all the tiles are accommodated in at most the maximum number of partitions. In each iteration, the unit partition size is first adjusted and then all the tiles are examined (in the already imposed SFC order) while performing the following operations:

1. See if the tile can fit in the same partition as the last tile without the number of objects exceeding the partition capacity.
2. If the addition of the new tile causes the last partition to overflow, start a new partition.

We illustrate the tile aggregation algorithm with an example. Given a set of 29 spatial objects and a threshold of at most 4 objects per tile, Phase 1 generates 13 tiles (see Figure 4(a)). The tiles are then numbered in the Hilbert SFC traversal order (see Figure 4(b)(left)). We assume that there are 4 nodes, N_1 to N_4, in the cluster. If each node gets an equal share of tiles (last node gets extra tiles) the tile-to-node allocation is:

$N_1 : T_1, T_2, T_3$[8 objects] $N_2 : T_4, T_5, T_6$[4 objects]
$N_3 : T_7, T_8, T_9$[8 objects] $N_4 : T_{10}, T_{11}, T_{12}, T_{13}$[9 objects]

Here, N_2 gets only 4 objects while N_4 gets 9 objects, giving a significant tuple distribution skew with this approach.

Next, we apply our tile aggregation algorithm with a partition size of 8. Traversing the tiles in SFC order, ptn_1 is assigned tiles T_1, T_2 and T_3. The next tile T_4 will not fit in ptn_1, so ptn_2 is created and filled similarly. The end result is:

$ptn_1 : T_1, T_2, T_3[8 \text{ objects}]$ $ptn_2 : T_4, T_5, T_6, T_7[7 \text{ objects}]$
$ptn_3 : T_8, T_9, T_{10}[7\text{objects}]$ $ptn_4 : T_{11}, T_{12}, T_{13}[7\text{objects}]$

Since the number of partitions is the same as the number of nodes, each node is assigned one partition. As can be seen, the data distribution skew is significantly reduced.

To enable dynamic load balancing, we actually generate many more partitions than the number of nodes.

3. DYNAMIC LOAD BALANCING

Load-balancing is essential to parallelize any computing task, including a long running analytics query. An imbalanced workload assignment may result in a significant delay between the completion times of the fastest and the slowest (aka *straggler*) machines. Even in a perfectly homogeneous cluster of machines, the inherent skew in the spatial dataset may cause workload imbalance. In the real world, machine performance can vary widely, as seen in Section 1.2, worsening the straggler effect.

Most traditional parallel database systems [9] do not deal with straggler nodes, as the underlying assumption is that the member nodes have identical performance. HadoopDB [1] deals with straggler nodes by launching speculative tasks, but its speculative task execution model (inherited from Hadoop) may lead to critical performance issues in heterogeneous environments.

We now describe Niharika's solution for load balancing, which does not rely on speculative tasks. We first discuss our dynamic partition assignment and then our scheduling algorithms.

3.1 Dynamic partition assignment

In a traditional parallel database, each member node is statically assigned a disjoint subset of the dataset. In the context of spatial query execution, this implies the static assignment of spatial partitions that are generated by the spatial declustering phase. When a query job is submitted, each node can be issued an identical query task, which it executes on its own dataset. The resultset returned by all member nodes can be aggregated to produce the final query result. For instance the query "find all the line segments that intersect polygon objects", can be expressed as follows in PostgreSQL:

```
select distinct a.gid from
  arealm_us a, edges_us e where
  ST_Intersects(e.the_geom, a.the_geom)   (1)
```

Each node executes this query on its data partition.

However, static partition assignment does not take into account node performance heterogeneity, dynamic load conditions, or the inherent processing skew due to variation in spatial object sizes and join selectivity. These factors can easily result in straggler nodes. Instead, we propose *dynamic partitioned parallelism* in which the data partitions that each node processes are determined "just in time" based on the performance of presently available nodes. This scheme requires that nodes either host the entire dataset locally, or obtain the dataset for their assigned partitions at run-time, since it is not known ahead of time which partitions will be needed. The compact nature of vector spatial data makes it tractable to host the entire dataset on each node, which is the approach we take. Designing a performance heterogeneity aware data placement algorithm, allowing a node to host a part of the whole dataset, is a future work.

Once the Scheduler has determined a partition assignment, we need to ensure that each node processes only the partitions that are mapped to it. We cannot send the original query to each node because this would result in each node processing the original query over the entire dataset. Instead, a form of virtual dynamic partitioning can be achieved by customizing queries on-the-fly to direct each node to process only its assigned partitions. This query customization scheme requires the addition of a partition_id column to the database tables. Then, the query can be tailored for each node by the Coordinator to include the appropriate partition identifiers.

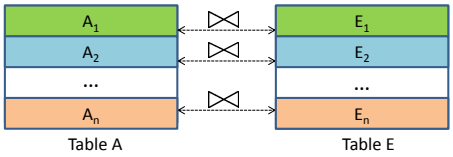


Figure 5: Spatial join with partitioned tables

Dynamic partitioning implies that the workload should be discretely divisible into chunks of almost equal size. Spatial partitions are the natural granularity for dividing spatial query workloads. The physical division of a table can be achieved by a widely supported database technique called *sharding*, which allows a single instance of a database table to be split into smaller physical pieces. PostgreSQL supports sharding by letting the sharded tables "inherit" from a master table and adding a constraint on the partitioning key [15]. In Niharika, the partitioning key is the spatial partition id and an index is created on this for each partition.

Query customization can be done in several ways. First, the **where** clause of the original query can be extended with a set of partition ids specified as a range (using **between**) or as a list (using **in**). We can also use a **union** of multiple queries, with each query specifying one partition. For example, we can specify the partitions for node N_2 from Section 2.2 by rewriting Query (1) using **union**:

```
...  and a.partition_id = T4
     and e.partition_id = T4
union select distinct a.gid from
   arealm_us a, edges_us e where
   ST_Intersects(e.the_geom, a.the_geom)
    and a.partition_id = T5
    and e.partition_id = T5
union ...
```

The choice of customization strategy depends on the underlying database engine. For instance, we want the query optimizer to select an ideal spatial join plan. Figure 5 shows a spatial join between partitioned tables A and E. An ideal plan would involve join between matching partitions only (e.g. partition 1 of table A would be joined only with partition 1 of table E). Joining partition 1 of table A with any other partition of table E is unnecessary, since it would not contribute any tuple to the query resultset. Other features of the database engine may affect the choice between query customization strategies that result in ideal query plans.

Inefficient query plan: $A \bowtie E = A_1 \bowtie E_1 U A_1 \bowtie E_2 ... U A_1 \bowtie E_n \cup A_2 \bowtie E_1 U A_2 \bowtie E_2 ... U A_2 \bowtie E_n ... \cup ... A_n \bowtie E_n$

Ideal query plan: $A \bowtie E = A_1 \bowtie E_1 \cup A_2 \bowtie E_2 ... \cup A_n \bowtie E_n$

We evaluated each of these customized queries for dynamic partitioning on PostgreSQL, and found that an ideal query plan was generated for only the **union** approach with sharded tables (*Sharding-union*). Unfortunately, we also found that memory usage ballooned with Sharding-union when a large number of partitions is specified (i.e. >250), leading to dramatic increases in execution time. To work around this limitation in PostgreSQL, we created a *Stored procedure* in which the Sharding-union query with multiple partitions is expressed as an iteration over each partition:

```
FOR p IN partition_list LOOP
    ...  and a.partition_id = $p
    and e.partition_id = $p ...
END LOOP
```

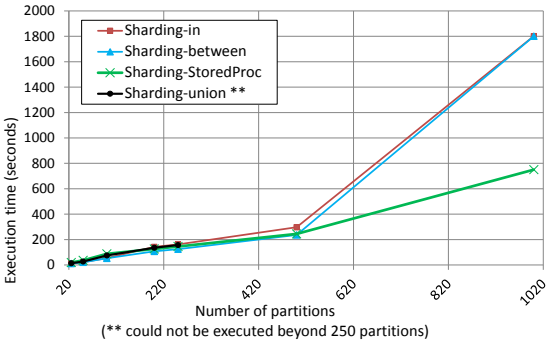


Figure 6: Execution times for LiAus query with different partition strategies

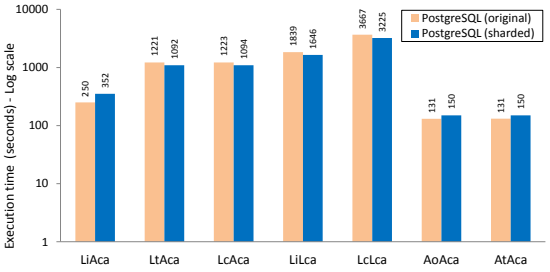


Figure 7: Query execution times - original vs Sharding-StoredProc (California dataset)

Figure 6 shows the effect of the different dynamic partitioning strategies on query execution time for LiAus on PostgreSQL as the number of partitions increases. The shared procedure approach (*Sharding-SharedProc*), gives us an ideal query plan that has stable performance; we use this strategy in our implementation.

Next we execute all the queries with Sharding-StoredProc approach on single PostgreSQL instances (results are averaged over the set of m1.xlarge nodes). The execution times are compared against those of the *original* PostgreSQL dataset (non-sharded). Figure 7 shows the comparison for California and Figure 8 for US dataset. In some cases the *original* PostgreSQL does better than *sharded* PostgreSQL, and in other case the reverse occurs. In all subsequent experiments we use the execution times of the single-node *sharded* PostgreSQL as the baseline to compute speedup for our scheduling and dynamic load-balancing strategies.

We now present several alternatives for partition assignment to achieve load balancing. These are evaluated in Section 4.

3.2 Multi-round assignment of partitions

An *a priori* balanced assignment of partitions to nodes in a single round is surprisingly difficult. Intuitively, such an assignment can be made by approximating the processing cost of each partition, and the processing capacity of each node, and allocating partitions to nodes proportionally. Node processing capacity can be approximated by running the same query against the same dataset on each, and comparing their relative performance.

We experimented with various strategies for estimating the node processing capacity and the work required per partition, for different queries, and then assigning partitions accordingly. The speedups achieved were inconsistent; we do not present the results due to lack of space. Heterogeneity-aware single assignment is limited because (a) it does not adapt to dynamic load conditions and (b) the work required per partition depends on the query being executed.

To overcome the limitations of single-round assignment, we need to adjust the size of the assigned workload as the query executes. This goal can be achieved by assigning work in multiple rounds,

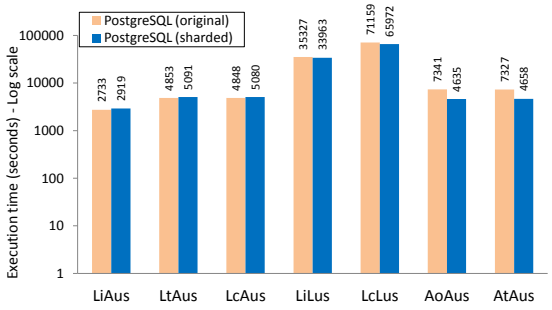


Figure 8: Query execution times - original vs Sharding-StoredProc (USA dataset)

Require: W is the number of DBWorkers; P is total number of partitions; $BatchSize$ is the size of assignment

```
1:  TotAssigned ← 0
2:  MaxIdx ← −1
3:  while true do
4:      for i = 0 to W − 1 do
5:          Status ← status of DBWorker i
6:          if Status = IDLE then
7:              if TotAssigned < P then
8:                  TotAssigned ← TotAssigned + BatchSize
9:                  if TotAssigned > P then
10:                     TotAssigned ← P
11:                 MinIdx ← MaxIdx + 1
12:                 MaxIdx ← MaxIdx + BatchSize
13:                 if MaxIdx > P − 1 then
14:                     MaxIdx ← P − 1
15:                 RwQuery ← Rewrite(MinIdx, MaxIdx)
16:                 Assign RwQuery to DBWorker i
17:                 Status ← ASSIGNED
18:                 AsgndCnt ← AsgndCnt + 1
19:         else if Status = COMPLETE then
20:             CmpltCount ← CmpltCount + 1
21:     if CmpltCnt = AsgndCnt and TotAssigned = P then
22:         break
```

Figure 9: Algo1: Multi-round fixed-size batch assignment

where a node processes a smaller portion of its overall assigned workload in each round. In effect, nodes that complete their work faster can be assigned a larger share of the total workload.

A useful abstraction for scheduling is Divisible Load Theory (DLT) [5]. It attempts to devise a task assignment plan such that all processors finish their computation simultaneously and any deviation from the optimal plan can be improved by transferring load from a busy processor to an idle one. We present three multi-round scheduling algorithms inspired by DLT.

3.2.1 *Fixed-size batch assignment*

The first algorithm is the multi-round fixed-size batch assignment to idle nodes (Algo1, see Figure 9). Initially, each node is assigned a batch of spatial partitions (generated in the declustering phase). Thereafter, any node that finishes processing its batch gets the next batch to work on. The intermediate results generated after processing each batch are combined and aggregated by the Coordinator. At each step the min and max partition indexes of the batch are updated (lines 11 through 14) and a query rewriting step (line 15) customizes the query before issuing it to the DBWorker. This process is repeated until all the partitions are completed and the final result is produced. Note that the list of partitions is traversed sequentially and any DBWorker can be assigned the next batch of partitions if it has finished processing its previous batch.

Intuitively, the choice of batch size requires a tradeoff between the overhead of executing the query in multiple rounds, and the opportunity to do load balancing. A smaller batch size requires more rounds, which creates more opportunities to adjust the workload assignment, but also incurs more overhead. It may be useful

```
Require: W, P, BatchSize as before; HistBucket[B][] is execution profile histogram having B buckets; InitAssigned[B][W] is a boolean array initialized false
1: while true do
2:   for i = 0 to W − 1 do
3:     CapacityRatio ← capacity ratio of DBWorker i
4:     Status ← status of DBWorker i
5:     if Status = IDLE then
6:       Assigned ← false
7:       for j = 0 to B − 1 do
8:         if InitAssigned[j][i] = false then
9:           AsgnList ← assign from HistBucket[j][]
10:            in proportion to CapacityRatio
11:          InitAssigned[j][i] ← true
12:          Assigned ← true
13:          break
14:      if Assigned = false then
15:        for j = 0 to B − 1 do
16:          if any unprocessed left in HistBucket[j][] then
17:            AsgnList ← assign from HistBucket[j][]
18:            Assigned ← true
19:            break
20:      if Assigned = true then
21:        RwQuery ← Rewrite(AsgnList)
22:        Assign RwQuery to DBWorker i
23:        Status ← ASSIGNED
24:        AsgndCnt ← AsgndCnt + 1
25:    else if Status = COMPLETE then
26:      CmpltCount ← CmpltCount + 1
27:  if CmpltCnt = AsgndCnt and TotAssigned = P then
28:    break
```

Figure 10: Algo2: Multi-round fixed-size batch assignment using node processing capacity

to employ different batch sizes for different nodes, however, we have found that a fixed-size batch for all nodes works well. Experimenting with variable batch sizes is left for future work. The main advantage of Algo1 is its simplicity and that there is no setup step.

Algo1 (Figure 9) is implemented as a Java program running in the Coordinator node. The Coordinator launches a worker thread for each DBWorker, which acts as a state machine. When the status is *IDLE* (line 6), the corresponding DBWorker is available for more work. When the status is *ASSIGNED* (line 17), the DBWorker executes the query assigned to it. When a DBWorker finishes its query and the result is available, the status is changed to *COMPLETE*. When all P partitions are processed, the final result is printed.

3.2.2 Batch assignment using processing capacity

In Algo1, node processing capacity is implicit, in the sense that more work is assigned to faster nodes because they become idle sooner. Our second algorithm (Algo2, described in Figure 10) explicitly considers node processing capacity. The idea is to assign batches of workload partitions to each node such that the total number of partitions assigned to each node is proportional to its processing capacity in the cluster. The capacity ratio of a node's performance can be based on the different types of nodes that are in the cluster (for instance, m1.xlarge, m2.xlarge, m2.2xlarge, etc). Another way to do this is by executing a representative query on each node and comparing their relative performance. To address skew in the dataset, the spatial partitions are grouped together in histogram buckets such that the partitions in the same bucket have similar execution times. The execution histogram is constructed during a preprocessing step, in which a representative query is executed on a single node for each partition and placing the partition id in the histogram based on its execution time.

While executing the query, the partitions are assigned from each bucket to nodes in proportion to their processing capacity (lines 7 through 13). The buckets with the longest running partitions are processed first. If a faster node completes processing all its share

```
Require: W, P as before; LargePartsList is a small list of partitions with the most skew
1: i ← 0
2: while LargePartsList has more items do
3:   append(NodeAsgnList[i], next(LargePartsList))
4:   i ← i + 1
5:   if i = W then
6:     i ← 0
7: Q ← P − size(LargePartsList)
8: PartIndex ← 0
9: for i = 0 to W − 1 do
10:  CapacityRatio ← capacity ratio of DBWorker i
11:  CurrAsgn ← Q ∗ CapacityRatio/TotalCapacity
12:  while PartIndex < Q and CurrAsgn > 0 do
13:    if PartIndex not in LargePartsList then
14:      append(NodeAsgnList[i], PartIndex)
15:      CurrAsgn ← CurrAsgn − 1
16:    PartIndex ← PartIndex + 1
17: returnNodeAsgnList
```

Figure 11: Procedure StrategicAssignment for Algo3

of partitions from all the buckets, it looks for yet unprocessed partitions from the buckets (lines 15 through 19). The query is customized with the assigned list of partitions (line 21) before the DBWorker executes the query.

3.2.3 Batch assignment with node affinity

The previous algorithms are agnostic to which node is assigned to process which partition. For relatively small datasets that can fit comfortably in the node memory, this should not be a concern, as all partitions can be cached in RAM. However, for larger datasets, an essentially random assignment of partitions to nodes can lead to very poor caching behavior. In this case, partitions will frequently need to be fetched from disk, since they are unlikely to be found in the memory of the node to which they are assigned.

Our third algorithm, Algo3, assigns partitions to each node from a preferred set of partitions, thereby maximizing the likelihood that the partitions are available in that node's memory. Of course, some spatial partitions may still need to be fetched from disk. However, Algo3 is expected to reduce disk I/O substantially compared to a random assignment such as Algo1.

The assignment approach of Algo3 (Procedure StrategicAssignment, Figure 11) first assigns to DBWorkers from a small list (typically less than 5% of total) of partitions that have the most skew. This ensures that the long running partitions will be the first to get processed. To assign from this list we do not consider node processing capacity, since this is a rather small set. Then from the remaining list of partitions each DBWorker is assigned a set of preferred partitions. This takes into account the static capacity ratio of a node's performance. The assignment algorithm makes sure that a node is always assigned the same set of partitions to maximize caching benefits. It also ensures that a node will have a sufficient number of partitions to process. We assume that a number of spatial queries are processed by the system. As outlined in lines 7 through 17 of Algo3 (Figure 12), each DBWorker processes from the list of partitions assigned to the node. However, if a DBWorker has processed all its assigned partitions, it may be assigned to work on any unprocessed partitions originally assigned to another node (lines 20 through 25). This allows Algo3 to deal with node performance heterogeneity and skew, while still providing good affinity between nodes and the partitions they usually process.

3.2.4 Round Robin with equal share of partitions

To evaluate the benefits of dynamic load balancing, we also implemented a single-assignment approach in which each node is given an equal number of partitions. We call this approach Round Robin with Equal Share of Partitions (RR-ESP). As the name sug-

```
Require: W, P, BatchSize as before; NodeAsgnList[W] is the list
    of partitions assigned to DBWorkers by Procedure StrategicAssign-
    ment; FetchList[W] is the list of partitions fetched by DBWorkers;
    PartAssigned[P] is a boolean array initialized to false
1:  while true do
2:    for i = 0 to W − 1 do
3:      Status ← status of DBWorker i
4:      if Status = IDLE then
5:        BatchList ← NULL
6:        if TotAssigned < P then
7:          CurrAsgnList ← NodeAsgnList[i]
8:          while CurrAsgnList has more items do
9:            if size(BatchList) = BatchSize then
10:             NodeAsgnList[i] ← CurrAsgnList
11:             break
12:           PartId ← next(CurrAsgnList)
13:           if PartAssigned[PartId] = false then
14:             PartAssigned[PartId] = true
15:             append(BatchList, PartId)
16:             remove(CurrAsgnList, PartId)
17:             TotAssigned ← TotAssigned + 1
18:         if size(BatchList) = 0 then
19:           for j = 0 to P − 1 do
20:             if PartAssigned[j] = false then
21:               PartAssigned[j] = true
22:               append(BatchList, j)
23:               TotAssigned ← TotAssigned + 1
24:               if size(BatchList) = BatchSize then
25:                 break
26:         if size(BatchList) > 0 then
27:           RwQuery ← Rewrite(BatchList)
28:           Assign RwQuery to DBWorker i
29:           Status ← ASSIGNED
30:     else if Status = COMPLETE then
31:       CmpltCount ← CmpltCount + 1
32:   if CmpltCnt = AsgndCnt and TotAssigned = P then
33:     break
```

Figure 12: Algo3: Multi-round batch assignment from a preferred partition set

Table 3: Database tables

Dataset	Database table	Geometry	Cardinality
California (4 GB, sharded)	edges_ca	polyline	4173498
	arealm_ca	polygon	7953
	areawater_ca	polygon	39334
Contig. USA (54 GB, sharded)	edges_us	polyline	73233790
	arealm_us	polygon	112492
	areawater_us	polygon	2290815

gests, it assigns the partitions to cores in a round robin fashion such that each node gets about the same number of partitions.

4. EXPERIMENTAL EVALUATION

In this section we evaluate our load balancing algorithms in various settings on the California and USA datasets. We first describe the datasets and the settings that we use in our experiments.

4.1 Experimental setup

We obtained the polyline (edges) and polygon (area landmass and area water) shapefiles for all the counties of California and created single shapefiles by merging them. Similarly, we created single US shapefiles for all the 48 states in the contiguous US. These shapefiles were then uploaded to a PostgreSQL database using the shp2pgsql tool. Table 3 outlines the database tables. To represent a real-world database system that processes many queries, we use the warm runs to calculate the speedup results. We observe that the overhead of duplicate elimination from resultset is minimal.

4.2 Results with dataset that fits in memory

Niharika's query execution model is naturally able to exploit multiple processing cores in each database node. PostgreSQL executes each query as a separate OS process, hence if multiple concurrent queries are issued to a multi-core node, the OS will naturally schedule them on different processors. Niharika's scheduler simply assigns multiple concurrent customized queries, each representing a separate batch of work, to each multi-core node. The number of concurrent batches is set to the number of cores in a node.

Algo1 and Algo2 were evaluated using Amazon EC2 with 4, 8, 12 and 16 **m1.xlarge** nodes, each of which had 4 cores and 15GiB memory. The speedup of each query is computed relative to the sharded single-node PostgreSQL execution time (as noted in Section 3.1) across all nodes. The speedup achieved by Algo1 and Algo2 for the seven queries is compared in Figure 13.

We observe that queries involving the same database tables have similar speedup profiles. For instance, the "Line and Area" queries generally have the lowest speedups for all numbers of cores, for both algorithms, while the "Line and Line" queries have the best speedups. The maximum achievable speedup is limited by the longest processing time of any single partition, and our analysis showed that large polygons in the area table led to some partitions with very long processing times in joins with the line table. We also observe that Algo2 has better speedup with these "Line and Area" queries than Algo1. However, with the other queries Algo2 performs worse than Algo1. Essentially, Algo2 tries to assign larger batches of work to the more powerful nodes in the cluster, thus reducing the number of rounds that are needed. When the estimates of node processing capacity and work per partition are reasonably accurate, Algo2 performs well. However, the larger batch size also means fewer opportunities for load balancing when conditions change dynamically, or when the static estimates are less accurate. Since Algo2 is more complex than Algo1, we do not use Algo2 for subsequent experiments.

4.3 Results with larger dataset (USA)

When the dataset is too large to remain in node memory, as is the case for the US dataset on m1.xlarge EC2 instances, we expect that Algo3 should have a significant advantage over the randomized assignment of Algo1. For comparison, we also include the single-assignment RR-ESP results in this section. Figure 14 shows the speedup attained by Algo1, Algo3 and RR-ESP against the best sharded single-node PostgreSQL performance. With Algo3, Niharika achieves excellent speedup in the number of cores for all queries, especially the "Line and Line" queries (LiLus and LcLus). For instance, the LcLus query achieves linear speedup in the number of cores, reaching 16.5X with 4 nodes (16 cores) and 62.8X with 16 nodes (64 cores). With "Line and Area" and "Area and Area" queries, Algo3 reaches the maximum achievable speedup, which is limited by the processing time of the longest running partition. For instance, with LtAus, Niharika attains speedup close to 30X with 8 nodes; beyond 8 nodes the speedup does not improve.

To explain this effect, we measured the processing time of each partition and found significant skew, as shown in Figure 15. The histogram (count) shows that only 1 partition took more than 150 (168.8s, allowing a maximum speedup of 30.1X), with 4 others taking between 100 and 150s. The remaining partitions took less than 80s. This result suggests that the spatial dataset properties can be a key constraining factor with Area queries. Although the TIGER dataset refactors long line objects, it does not do as well with large Area objects, as it does with the Line objects. To illustrate, we removed the largest 50 objects (0.4% of the dataset) from the arealm_us dataset and re-ran the Line and Area queries with Algo1, Algo3, RR-ESP and the sharded single-node PostgreSQL. As can be seen in Figure 16, Algo3 achieves near linear speedup in the number of cores. As future work, we are exploring a more

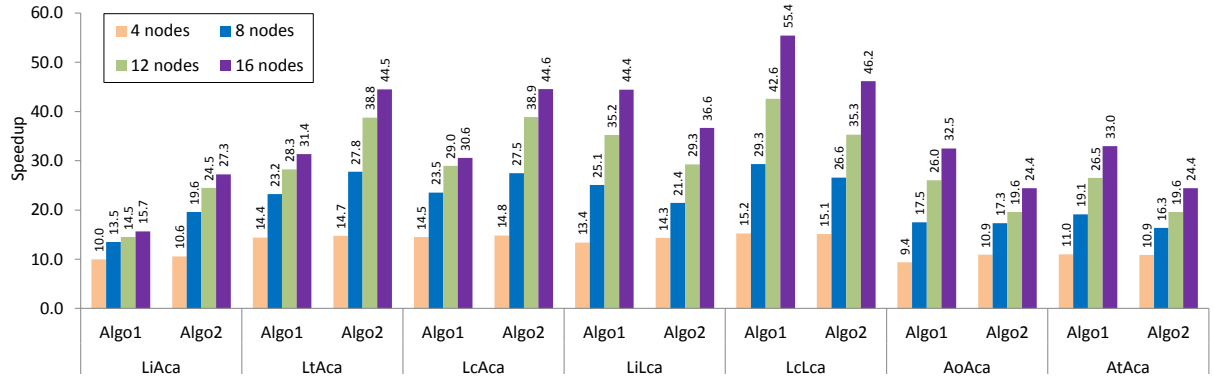


Figure 13: Speedup of Algo1 compared with Algo2 (Cal. dataset, 4 cores/node)

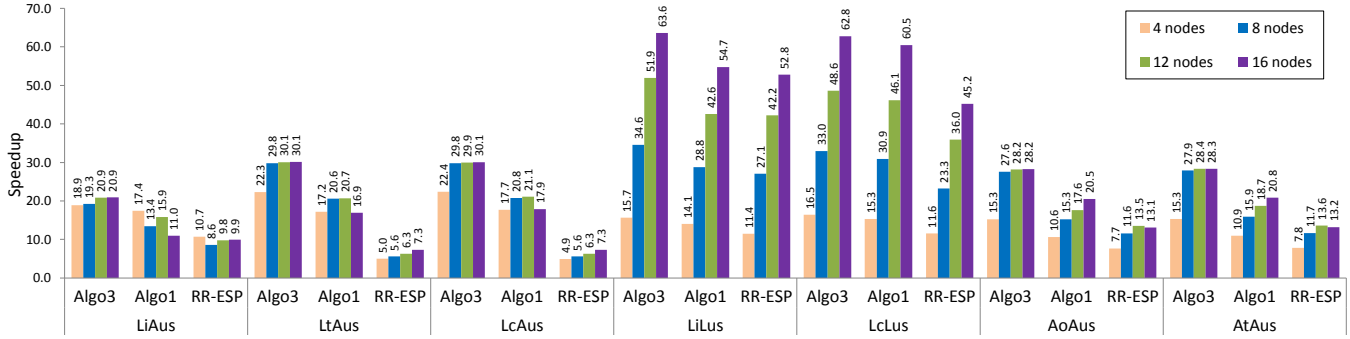


Figure 14: Algo3 vs Algo1 vs RR-ESP speedup (USA dataset, 4 cores/node)

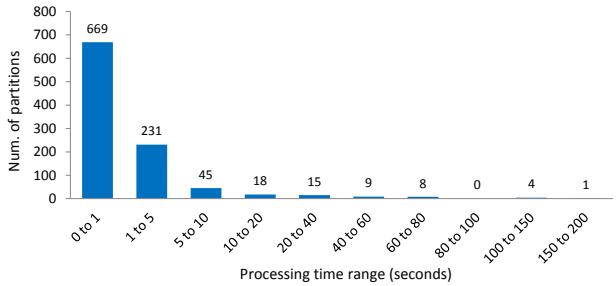


Figure 15: Histogram of the processing times of the partitions for LtAus (USA dataset, Algo3)

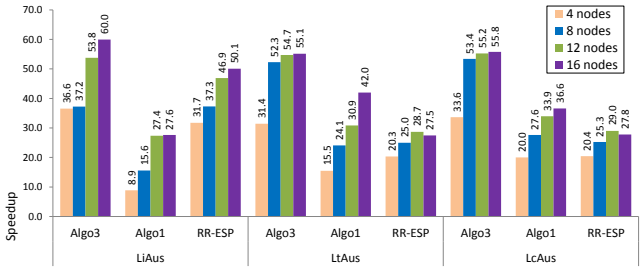


Figure 16: Algo3 vs Algo1 vs RR-ESP speedup with modified dataset (USA dataset, 4 cores/node)

fine-grained declustering approach that takes object size as well as object count into consideration.

In comparison to Algo3, the speedup of Algo1 is significantly worse on all queries. Although Algo1 does a good job with in-memory datasets, that is no longer the case with datasets that do not fit in memory. Therefore, Algo3 can be the considered the load balancing algorithm of choice among the ones we present.

We also see that Niharika with Algo3 is superior to an approach that is agnostic to performance heterogeneity, that is, RR-ESP. In both Figures 14 and 16 RR-ESP showed much poorer speedups compared to those of Algo3. Also, RR-ESP shows reduced speedups when going from 48 to 64 cores for all but one case (LiAus, where the speedup is unchanged) when the queries involve an area table. This suggests that RR-ESP is more susceptible to processing skew.

5. RELATED WORK

Spatial join is a highly complex database operation that is important in many spatial applications. A number of projects explored various ways to improve its performance, as surveyed by Jacox and Samet [12]. But, only a few have studied parallelizing spatial join.

Brinkhoff et. al [7] presented an R*-tree index based spatial join in a shared-virtual-memory architecture. They parallelize the filter step by assigning subtrees of the index to each processor. Zhou et al. [24] presented a spatial join algorithm based on static partitioning. However, their algorithm requires the dataset to fit in memory, and the system was evaluated with only one query. A non-blocking spatial join algorithm was proposed by Luo et al. [13]. However, their goal was to quickly produce the first result tuple, rather than completing the query execution. Two partitioning-based parallel spatial join algorithms, clone join and shadow join, were presented in [17]. Individual partitions are joined using the PBSM algorithm [16], which uses a plane sweeping technique. This approach works best when there is no index on the spatial attribute. In contrast, Niharika takes advantage of R-tree index on each partition. Their techniques also do not account for node heterogeneity and so suffer from performance issues due to "straggler" nodes.

The previous parallel join approaches appeared prior to the multicore era. The abundance of processing cores brings new opportunities as well as new challenges, such as the lack of support for intra-query parallelism in many well-established databases, including PostgreSQL. Recently, several projects [2, 4] have studied intra-query parallelism for the emerging multicore machines but their focus is primarily on traditional database workloads. Spatial

join has not received much attention, although it offers great potential for parallelism due to its compute intensive nature [20].

The emergence of the Cloud computing brings additional challenges to distributed query processing. Performance heterogeneity is a norm [10], rather than an exception, in these platforms. Addressing performance heterogeneity in data processing frameworks for Cloud, such as MapReduce, is a very active area of research [22]. Although Mayr et al. [14] recognized potential issues with parallel query processing in a heterogeneous cluster, it has since received very little attention from the database community.

SJMR [23] is system that parallelizes spatial join operations using MapReduce on clusters of commodity machines. It does not leverage the features of an underlying RDBMS, such as spatial indexes and buffer cache management, and instead uses Hadoop to process vector data stored in a simple string format. SJMR shows a slight reduction in execution time over a parallel implementation of PBSM, when joining 2 tables from the TIGER/Line for California.

Recently, Aji et al. produced a MapReduce-based system for medical image processing, which also focuses on spatial join [3]. They also observed issues with skew in the dataset. Unfortunately, a performance comparison is not possible, since they use a proprietary dataset and the code is not available. Moreover, results are reported starting at 20 reducer nodes, so speedup and parallel efficiency against a sequential implementation cannot be calculated.

It has been demonstrated [18] that MapReduce systems such as Hadoop are significantly slower than parallel databases. Thus, it is beneficial to take advantage of the database query processing techniques, including spatial indexes. Niharika is inspired by HadoopDB [1], as it utilizes individual PostgreSQL instances in each member node. However, HadoopDB does not support spatial data types and spatial query execution. Niharika differs from HadoopDB architecturally and does not have Hadoop's overhead. Niharika's heterogeneity-awareness and multi-round query execution model are unique among parallel query processing systems.

6. CONCLUSIONS

Spatial join is at the heart of many emerging spatial data analysis applications. We have introduced Niharika, a parallel spatial query execution infrastructure, designed to exploit multiple cores in modern processors to accelerate spatial join performance.

Performance heterogeneity in a cluster is natural in Cloud computing settings. We have also shown that even an apparently homogeneous cluster can have significant performance heterogeneity, which hurts parallel database query execution times. With its load-balancing techniques Niharika is able to dynamically perform "performance proportional" assignment of workloads to nodes according to their processing capacity. Niharika significantly reduces idle times for faster nodes in relation to a straggler.

Niharika does not require changes to the database engine. Its query execution model can naturally use all available processing cores for intra-query parallelism, which is not currently supported by many databases including PostgreSQL. We presented three multi-round load-balancing algorithms and showed the importance of dynamically adapting to performance heterogeneity. Niharika's Algo3 scales well with an in-memory dataset and a dataset that does not fit in memory on a cluster of multicore nodes in the Cloud.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This research was supported by an NSERC Discovery Grant and an Amazon AWS in Education grant. Suprio Ray is supported by an NSERC PGS-D scholarship.

8. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, pages 922–933, 2009.

[2] R. Acker, C. Roth, and R. Bayer. Parallel query processing in databases on multicore architectures. In *ICA3PP*, 2008.

[3] A. Aji, F. Wang, and J. H. Saltz. Towards building a high performance spatial query system for large scale medical imaging data. In *SIGSPATIAL*, pages 309–318, 2012.

[4] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. In *VLDB*, pages 1064–1075, 2012.

[5] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. Scheduling Divisible Loads in Parallel and Distributed Systems. *IEEE Computer Society*, 1996.

[6] D. Borthakur. Petabyte scale databases and storage systems deployed at facebook. In *SIGMOD*, 2013.

[7] T. Brinkhoff, H. peter Kriegel, and B. Seeger. Parallel Processing of Spatial Joins Using R-trees. In *ICDE*, 1996.

[8] P. C. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree load balancing using space-filling curves. *Williams College, TR CS-03-01*, 2003.

[9] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Commun. of the ACM*, 35(6):85–98, 1992.

[10] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift. More for your money: exploiting performance heterogeneity in public clouds. In *SoCC*, 2012.

[11] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 79–87, 2007.

[12] E. H. Jacox and H. Samet. Spatial join techniques. *ACM Transactions on Database Systems*, 32(1), 2007.

[13] G. Luo, J. F. Naughton, and C. J. Ellmann. A Non-Blocking Parallel Spatial Join Algorithm. In *ICDE*, 2002.

[14] T. Mayr, P. Bonnet, and J. Gehrke. Leveraging non-uniform resources for parallel query processing. In *CCGrid*, 2002.

[15] PostgreSQL Partitioning. http://www.postgresql.org/-docs/8.3/static/ddl-partitioning.html.

[16] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[17] J. M. Patel and D. J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *SIGSPATIAL*, 2000.

[18] 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*, 2009.

[19] S. Ray, B. Simion, and A. D. Brown. Jackpine: A benchmark to evaluate spatial database performance. In *ICDE*, 2011.

[20] B. Simion, S. Ray, and A. D. Brown. Surveying the landscape: An in-depth analysis of spatial database workloads. In *SIGSPATIAL*, pages 376–385, 2012.

[21] http://www.census.gov/geo/www/tiger.

[22] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.

[23] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *CLUSTER*, pages 1–8, 2009.

[24] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, 1998.