Skew-Resistant Parallel In-memory Spatial Join

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 join is a crucial operation in many spatial analysis applications in scientific and geographical information systems. Due to the compute-intensive nature of spatial predicate evaluation, spatial join queries can be slow even with a moderate sized dataset. Efficient parallelization of spatial join is therefore essential to achieve acceptable performance for many spatial applications. Technological trends, including the rising core count and increasingly large main memory, hold great promise in this regard. Previous parallel spatial join approaches tried to partition the dataset so that the number of spatial objects in each partition was as equal as possible. They also focused only on the filter step. However, when the more compute-intensive refinement step is included, significant processing skew may arise due to the uneven size of the objects. This processing skew significantly limits the achievable parallel performance of the spatial join queries, as the longest-running spatial partition determines the overall query execution time.

Our solution is SPINOJA, a skew-resistant parallel in-memory spatial join infrastructure. SPINOJA introduces MOD-Quadtree declustering, which partitions the spatial dataset such that the amount of computation demanded by each partition is equalized and the processing skew is minimized. We compare three *work metrics* used to create the partitions and three load-balancing strategies to assign the partitions to multiple cores. SPINOJA uses an in-memory column-store to store the spatial tables. Our evaluation shows that SPINOJA outperforms in-memory implementations of previous spatial join approaches by a significant margin and a recently proposed in-memory spatial join algorithm by an order of magnitude.

1. INTRODUCTION

Spatial join is pivotal in many scientific and geospatial applications such as cartography, city planning, land surveys and astrophysics simulations. With the rapid rise in the volume and variety of spatial data, "spatial analytics" is no longer just a niche in scientific organizations and academia. The popularity of on-line services are also contributing to the growth of spatial analysis applications. For instance, potential property buyers may be interested in services that offer information about flood-risk or the danger of a toxic spill nearby. These online applications must respond to user queries within a reasonable amount of time. Novel sources of digital spatial datasets are also helping to create new spatial analysis applications. For instance, recent innovations in digital pathology slide scanner technology have radically altered anatomic pathology. This is paving the way to the quick and affordable generation of large quantities of spatially derived micro-anatomic data. Systematic analysis of such data involves complex spatial join queries [2] and due to the critical nature of the application domain, fast query response time is highly desirable.

Spatial queries typically 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 indexing, leading to very long query latencies. To reduce this cost, query evaluation consists of a filter step that eliminates as many objects as possible based on an approximation of the object geometry, and a more costly refinement step that uses the actual geometries of the remaining candidate objects. In spite of this optimization, spatial join queries remain compute-intensive with long latencies.

Several prior research projects have investigated efficient algorithms to address this important spatial join problem, with a focus on the filter step. Many of these approaches [2, 15, 16, 19, 21] rely on spatial declustering to partition the dataset, enabling parallelization. These techniques are based on creating spatial partitions or tiles such that each tile has roughly the same number of objects. However, when both the filter and refinement steps are considered, significant processing skew may be observed due to the variation in the time it takes to process different tiles. This variation results from differences in object properties, such as size or point density, which are typical in many spatial datasets such as geospatial and VLSI. To improve parallel performance of spatial join, the issue of processing skew must be addressed.

The rapid growth in core counts and main memory sizes presents significant opportunities for improving the performance of spatial join queries. Today's high-capacity servers may have hundreds of cores and up to a terabyte of main memory. Due to the compactness of the vector data representation, many spatial datasets can fit completely in the main memory of a more modest machine. For instance the TIGER [20] dataset, comprised of the polyline and polygon features of all the contiguous states of the USA, is roughly 54 GB in size [19]. To exploit the potential of large memory machines, Nobari et al. recently proposed TOUCH, an in-memory spatial join algorithm [12]. Like previous spatial join approaches, TOUCH focuses on the filter step and does not address the processing skew inherent in many datasets and spatial analysis applications.

We introduce SPINOJA (Skew-resistant Parallel IN-memOry spatial Join Architecture), a main memory query execution infrastruc-

SSDBM '14, June 30 - July 02 2014, Aalborg, Denmark.
Copyright 2014 ACM 978-1-4503-2722-0/14/06...$15.00.
http://dx.doi.org/10.1145/2618243.2618262

ture designed specifically to address processing skew for parallel in-memory spatial join. SPINOJA takes into account the overall query execution time, including the filter and refinement steps. It introduces a new spatial declustering scheme called MOD-Quadtree declustering, which alleviates processing skew. Unlike previous declustering approaches that try to distribute the number of objects evenly, our approach attempts to equalize the amount of required computation per partition. The main idea is to decompose objects along tile (spatial partition) boundaries such that the amount of work involved in processing the tiles is roughly equal. We explore three work metrics to determine the best way to approximate the amount of work per tile. We also present three load-balancing strategies to assign the tiles to worker threads.

To evaluate SPINOJA fairly against prior work, we implement parallel in-memory versions of previously proposed disk-based spatial join approaches, in addition to a parallel version of the TOUCH in-memory spatial join approach. We evaluate eight spatial join queries involving five spatial predicates that require complex refinement processing. Our experimental results show that SPINOJA does a very good job of minimizing the processing skew and achieves superior performance over the other approaches. In addition, the memory requirements of SPINOJA are much lower than those of Clone Join and comparable to those of TOUCH.

Our contributions are as follows:

1. We experimentally demonstrate that processing skew is a significant bottleneck to the parallel performance of spatial join, particularly when both the filter and refinement steps are taken into account.
2. We present a novel spatial declustering technique based on work metric and several load-balancing strategies to address processing skew and significantly improve overall query performance.

The rest of the paper is organized as follows. Related works are discussed in Section 2.2. We describe the motivation of our work in Section 3. Then we introduce the overall system organization and the details of SPINOJA in Section 4. The evaluation is presented in Section 5 and we draw conclusions in Section 6.

2. BACKGROUND AND RELATED WORK

In this section we provide some background on spatial query execution. Next we present previous research related to spatial join.

2.1 Spatial queries

A spatial query execution system is essentially a relational database system with enhanced support for spatial data types, spatial indexing and spatial join [9]. Geometric objects are modeled by spatial data types such as point, line and polygon. Topological relations are used to specify how the geometric objects are associated in a two dimensional space. Some of the common relationships, also called spatial predicates, are *Intersects*, *Overlaps*, *Touches*, *Equals*, *Contains* and *Disjoint*. Recently, the Open Geospatial Consortium (OGC) defined a core set of operations [13].

Most spatial applications today use vector data to represent complex map features such as parks (represented as polygons) or roads and rivers (represented by polylines). Vector data is much more compact than its raster data counterpart. Vector data also enables the use of sophisticated computational geometry algorithms for common spatial operations such as shape intersection.

Spatial query execution uses a two step evaluation mechanism. The first step is *filter*, which returns a superset of the candidate objects satisfying a spatial predicate, by comparing an approximation of actual objects (called the minimum bounding rectangle, or MBR). The second step is *refinement*, in which the actual geometries of the candidate objects are inspected. The filter step tries to eliminate as many objects as possible, since the refinement step uses time-consuming compute-intensive computational geometry algorithms.

2.2 Related work

Jacox and Samet present a comprehensive survey of various spatial join techniques [10], which they classify into two main categories: external memory and internal memory methods. We adopt their classification for spatial joins in our discussion. Due to the relevance to our work, we also present projects related to parallel spatial join separately.

2.2.1 Spatial Join

The external memory or disk-based approaches attempt to minimize the disk I/O involved in fetching the actual objects from the disk. The two step processing of spatial join, involving filter and refinement, was introduced by Orenstein [14] so that only those objects that satisfy the filter step needed to be fetched from disk. The filter step uses MBRs, which can be used to build a spatial index, such as an R-tree. When a spatial index exists on one of the datasets A, then an indexed nested loop join based technique can be used. With this approach, the dataset B is iterated over and for each entry $b \epsilon B$ the index on A is traversed to perform a window search based on the MBR of b. However, if the cardinality of dataset B is much higher than that of A, this approach is not very efficient.

When there is no pre-existing index on either of the datasets, partitioning-based join techniques can be used. The main idea is to decompose the spatial universe into partitions and then perform pairwise spatial join for each partition. Because an object may overlap multiple partitions, the result-set must be processed to eliminate duplicates. PBSM (Partition Based Spatial-Merge) [15] uses this strategy by decomposing the spatial domain into equal sized tiles. In the filter step for each tuple in a relation the <MBR, OID> pair is appended into a disk file. Then these key-pointer elements from the relations are loaded into memory for one single partition at a time and MBR-join is performed using plane-sweep. The resultant candidate set consists of OID pairs from both the relations. The candidate set is sorted and the tuples (i.e. actual geometry objects) are read sequentially into memory to perform the refinement step on the join attributes.

When the dataset completely fits in memory, the disk-based approaches can be used to perform in-memory spatial join. However, a technique specifically designed to perform in-memory spatial join may perform better than others. Without many disk-based restrictions, such as R-tree node size, there is more freedom in choosing various design parameters. These insights led Nobari et al. to develop an in-memory spatial join called TOUCH [12], in the context of computational neuroscience simulation. TOUCH creates an R-tree index on one of the relations. The key idea is to directly assign the objects of the other relation to the index of the first to a non-leaf level where they are fully contained by an MBR. This serves to filter out many object pairs from the candidate set, which are then joined using a PBSM-like partition join. TOUCH was shown to have better single-threaded performance over the in-memory implementations of several spatial join approaches. A limitation of TOUCH is that its index works for a particular pair of relations, because of the need to assign the objects of second relation to the R-tree index of the first relation. If there are more than two spatial relations in a dataset, a separate R-tree construction and object assignment must be performed for each pair of relations.

2.2.2 *Parallel Spatial Join*

Parallelization has been studied by a few previous works as a means to address the performance issues with spatial join processing. When both datasets are indexed by the same indexing technique such as R-tree (or variations) it is possible to use both indexes simultaneously. Brinkhoff et al. [5] presented such an approach. Targeted for shared-virtual-memory architecture, their R*-tree index based spatial join parallelizes the filter step by assigning subtrees of the index to each processor. Starting at the root, the R-tree indexes are synchronously traversed down to the leaf level. If two nodes from two trees at the same level intersect, then their children are checked for intersection.

Many disk-based parallel spatial join approaches [2, 16, 19, 21, 22] use spatial declustering to subdivide the spatial domain into *tiles* so that different tiles can be concurrently processed by different processors. The declustering phase attempts to reduce the variation in the number of objects in different tiles (known as tuple distribution skew) by evenly distributing objects to tiles. However, due to the properties of spatial datasets, such as object size and point density, different objects may require different amounts of computation in the refinement step. Thus, even when the number of objects per tile is about the same, *processing skew* may result. This is detrimental to the performance of a disk-based parallel spatial join algorithm.

Zhou et al. [22] presented an early grid partitioning based parallel spatial join approach. The paper addressed workload balancing to deal with skew in the parallel filter stages. Their system was evaluated with only one query.

Patel and DeWitt extended their PBSM work to deal with parallel spatial join [16]. Two partitioning-based parallel spatial join algorithms, Clone Join and Shadow Join, were proposed. Clone Join relies on a declustering technique similar to PBSM, called D-W (decluster with whole tuple replication). As the same suggests, entire tuples are replicated or cloned to nodes whose MBRs overlap with the tiles mapped to those nodes. Clone Join creates a large number of disjoint tiles and then maps them to virtual nodes in a round robin fashion. This scheme is expected to reduce tuple distribution skew and hence balance the load, since nearby tiles, having similar densities of objects, are assigned to different processors. After both relations are declustered using D-W, they are locally joined at each node using PBSM. Finally, the duplicates are eliminated from the result using a distinct operator. Shadow Join uses a different declustering algorithm called Partial Spatial Surrogates (PSS). Each tuple has a designated home node where the complete tuple is stored. In addition, a *fragment box* is represents the parts of the MBR of that tuple that overlap the tiles covered by any other node. The fragment box and the OID of each tuple (together called PSS) are then shipped to the destination node. Both relations are declustered using PSS and the candidate set is produced by joining partial spatial surrogates. Next, a redeclustering step sends the candidates to their home nodes where they are joined with the first relation. Then another redeclustering step sends the intermediate result set to their home nodes to join with the second relation. The authors reported that Clone Join performed better than Shadow Join as the join selectivity increased.

The emergence of distributed data processing frameworks, such as MapReduce, offered a new opportunity to parallelize spatial query processing. A few systems were developed based on this idea. SJMR [21] was designed to parallelize spatial join operations on clusters of commodity machines using Hadoop. They used a grid-based splitting approach with a Z-curve tile coding method and a round-robin tile-to-partition mapping scheme. When joining 2 tables from the TIGER/Line dataset for California, SJMR showed slightly worse performance than a parallel implementation of PBSM.

Table 1: Jackpine queries and abbreviations

Description (tables involved)	Abbreviations
Polygon overlaps Polygon (Areawater and Areawater)	Aw_ov_Aw
Polygon overlaps Polygon (Areawater and Arealm)	Aw_ov_Al
Polygon within Polygon (Arealm and Areawater)	Al_wi_Aw
Polyline Crosses Polygon (Edges and Arealm)	Ed_cr_Al
Polyline Intersects Polygon (Edges and Areawater)	Ed_in_Aw
Polyline Crosses Polygon (Edges and Areawater)	Ed_cr_Aw
Polyline Touches Polygon (Edges and Areawater)	Ed_to_Aw
Polyline Crosses Polyline (Edges and Edges)	Ed_cr_Ed

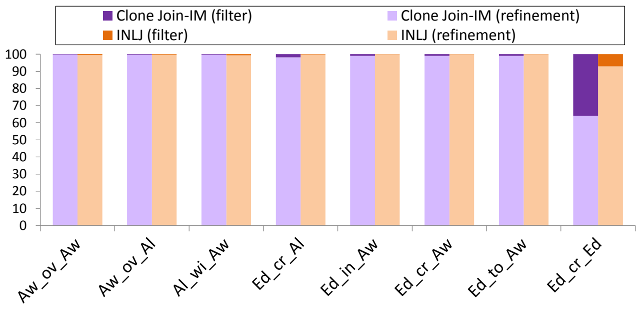

Figure 1: % execution time spent in filter vs. refinement with Clone Join-IM and INLJ

A MapReduce-based data warehousing system for medical image processing was developed by Aji et al. [2], which also utilizes spatial join. They used spatial partitioning and observed issues with skew in their dataset. To address this they use object count threshold as the metric to split each tile into two. Objects crossing multiple tiles are replicated to those tiles. Their query execution engine supports on-demand R*-tree index construction on a per tile basis.

We previously introduced Niharika [19], a parallel spatial query system that addresses the node performance heterogeneity in the Cloud during the execution of parallel spatial join queries. Like previous approaches, its spatial declustering scheme attempts to evenly distribute the number of objects to each partition. These spatial partitions are imported into PostgreSQL database tables as "shards". Niharika parallelizes spatial join by using individual PostgreSQL instances in each member node and assigning the shards to each node in a multi-round query execution model. We also observed processing skew in Niharika, particularly with queries involving polygons. In this paper, we show that the magnitude of the skew issue is even more prominent with in-memory spatial join.

3. MOTIVATION

Spatial join queries are used to coalesce two different datasets based on a spatial predicate. They are vital in a wide range of spatial analysis applications, from scientific simulations, cartography, and land surveys to flood risk analysis. New classes of spatial analysis applications are also emerging, in domains as diverse as VLSI, building information management, water/gas utilities, medical imaging, digital pathology and computational neuroscience.

As described in Section 2.1, spatial join queries use a two-step evaluation process. The refinement step typically involves computational geometry algorithms to evaluate the relationships between spatial objects, which impose a high computational load, even with spatial indexes. Consequently, a spatial join query involving even a moderately sized dataset may lead to long query latencies. Previous spatial join research efforts focused on the filter step and did not consider refinement (e.g., [4, 11, 15]). Moreover, they only considered one spatial predicate based on MBR comparison (MBR over-

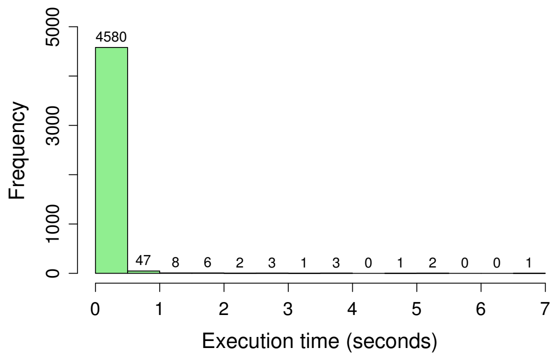

Figure 2: Histogram of the execution time per spatial partition (tile) for query Ed_cr_Al

laps or intersects). Since the refinement step dominates the overall query latency, we argue that it is important to consider both the filter and refinement steps. To illustrate our point, we selected 8 spatial join queries from the Jackpine spatial database benchmark [18]. These queries are shown in Table 1. To identify the specific tables involved in the join we use abbreviations with the 2 initial letters of the tables. We also selected five different spatial predicates defined by OGC. In Figure 1 we show the percentage of execution time spent in filter and refinement steps for these queries. We implemented in-memory versions of two disk based spatial join approaches: Clone Join and Indexed Nested Loop join (INLJ) based on in-memory R-tree filtering. We use the label **Clone Join-IM** to distinguish the in-memory version of Clone Join. We use the TIGER California dataset, details of which are in Table 3 in Section 5.1. As Figure 1 shows, for all queries, except for Ed_cr_Ed, the refinement step takes over 98% of the overall filter+refinement time. Therefore, it is imperative that a spatial join approach be optimized for refinement to address the long query latencies.

The recent TOUCH [12] in-memory spatial join approach, like previous disk-based approaches, also only considers the filter step for the evaluation. With an in-memory spatial join, disk latency is no longer an issue, and the refinement step dominates the overall query latency. When the actual object geometries are used during refinement, the processing skew can become a critical issue.

To illustrate the problem, we use the Clone Join-IM algorithm. We execute the Ed_cr_Al query for each tile generated by Clone Join-IM declustering and report the execution times in a histogram in Figure 2. As can be seen, the vast majority of the tiles took less than 0.5 seconds. Only three tiles took more than 5 seconds, and only one took over 6 seconds, which indicates significant processing skew. In a parallel spatial join query processing system it is important to distribute equal amounts of work to each processor. If a processor takes significantly longer than others due to processing skew, it becomes a "straggler" and the overall query latency suffers.

4. SPINOJA

We now describe various aspects of our system, called SPINOJA. SPINOJA is an in-memory parallel query execution system developed with the goal of minimizing the spatial join query latency. It addresses the challenges of processing skew using a metric-based spatial declustering scheme and dynamic load balancing.

4.1 System organization

SPINOJA's system organization is modeled after the master-slave architecture. As shown in Figure 3, the task scheduler is called the *TaskManager*, and worker threads are called *Workers*. The TaskManager is responsible for scheduling a query job and assigning parts of each job or tasks to the available Workers. The TaskManager also performs result aggregation, which involves spatial predicate evaluation and group-by operations, and eliminates duplicates from the resultset using a distinct operator.

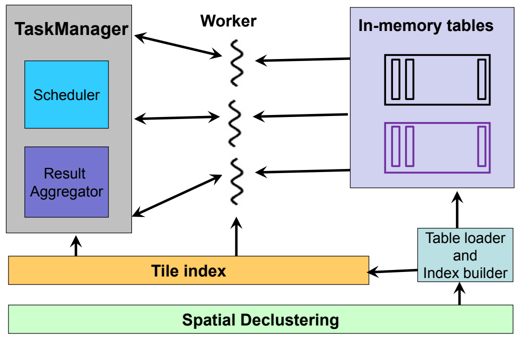

Figure 3: SPINOJA system organization

SPINOJA uses a spatial declustering scheme (described next in Section 4.2) that attempts to reduce processing skew. Key to this declustering process is the work metric and we describe three such metrics. The *Table loader and Index builder* component loads the tiles created by the spatial declustering into in-memory tables. It also constructs the *Tile index* to be used during the query execution. To implement the Tile index we use a compressed bitmap, which requires less memory than other data structures. More specifically, we use the CONCISE [7] compressed bitmap. The Scheduler component of TaskManager assigns the tiles to the workers using load-balancing algorithms such that each Worker only processes objects from its assigned tiles. We evaluate three algorithms for dynamic load balancing within this framework.

4.2 MOD-Quadtree declustering

A join query can be parallelized by partitioning the data domain into disjoint chunks and assigning the corresponding pair of chunks from the two tables to a separate processor. This spatial declustering process involves subdividing the spatial domain into 2-dimensional *tiles*. Each tile contains those objects whose MBRs it overlaps, which implies that any object whose MBR is overlapped by multiple tiles may need to be replicated to each of those tiles. As noted in Section 2.2.2, previous approaches to spatial declustering aimed to equalize the number of objects in each tile to try to ensure that each tile would incur roughly the same amount of computation. This may work if only the filter step is considered, as it makes only four numeric comparisons to check if the MBRs of two objects overlap. However, when the refinement step is also taken into account, this approach does not work because the amount of computation is dependent on the properties of the objects and the underlying computational geometric algorithm. Moreover, the replication of large objects to multiple tiles increases the amount of computation for each such tile.

To address the issues with existing spatial declustering approaches, SPINOJA uses a novel technique we call MOD-Quadtree (metric-based object decomposition quadtree) declustering. The main idea behind our technique is to create the tiles such that the amount of computation required by each tile is roughly the same. As the name suggests, MOD-Quadtree is a region quadtree variant that recursively decomposes a tile into four equal-sized tiles. The decomposition criteria is not based on the number of objects, but rather on the amount of computation estimated by a suitable *work metric*. When the amount of computation among the tiles is equalized,

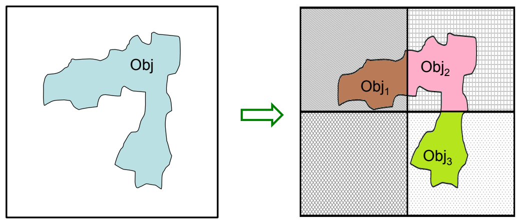

Figure 4: MOD-Quadtree object decomposition

```
Require: tileMBR is the MBR of the tile for which the metric is
    calculated; tableList is a list of tables containing the original
    spatial objects
1: workMetric ← 0
2: for tab in tableList do
3:     origObjList ← retrieveOverlappingObjects(tileMBR, tab)
4:     for obj in origObjList do
5:         objFrag ← getIntersection(obj,tileMBR)
6:         workMetric ← workMetric + calculateMetric(objFrag)
7:         . . .
8: return workMetric
```

Figure 5: Procedure getWorkMetric4Tile

it is expected that it would take roughly the same amount of time to process each tile and that there would be no “straggler effect”. A suitable work metric is therefore important to evenly distribute the work among the tiles. Another key aspect of MOD-Quadtree is that when a tile is subdivided into four, any object that overlaps with the newly created (smaller) tiles is also decomposed along the boundaries of them. This is illustrated in Figure 4, in which an object *Obj* is decomposed into three fragments Obj_1, Obj_2 and Obj_3 by clipping against the four tile boundaries. The termination criteria of quadtree approaches is typically related to a threshold on the number of objects, for instance, with point region quadtree the decomposition of a tile halts when a tile contains one point. In MOD-Quadtree, the recursive decomposition stops when the total number of tiles created so far exceeds a threshold.

In each round of the MOD-Quadtree declustering algorithm the tile with the largest work metric value is selected for decomposition. The center point of its MBR is calculated before removing the tile from a global tile list. A new tile *tile0* is created and its properties are set, in particular, the work metric is calculated for the newly created tile by invoking the procedure *getWorkMetric4Tile*. Then *tile0* is inserted into the global tile list. Similarly, tiles *tile1*, *tile2* and *tile3* are created and inserted into the global tile list. The MOD-Quadtree declustering algorithm is parallelized to attain multi-threaded performance and reduce overall execution time. Note that the tiles are ordered with Hilbert SFC (space-filling curve) orientation to enable traversing them using Hilbert order when assigning to Workers. This allows nearby tiles with similar object densities to be assigned to different Workers during query execution. Hilbert ordering also prevents a drawback with tile assignment approaches such as round robin or range partitioning. With round robin assignment, the tiles assigned to a Worker may form long columns when the number of tiles are integral multiples of the number of Workers. This may degrade overall performance, as was observed in [15]. With range assignment large blocks of tiles may be formed and also degrade performance.

Figure 5 shows the *getWorkMetric4Tile* procedure. It iterates through each spatial object that overlaps the MBR of a newly created tile *tileMBR* and determines the object fragment (line 5). Then the work metric is calculated and a local variable is updated (line 6). These steps are repeated for each of the tables involved in

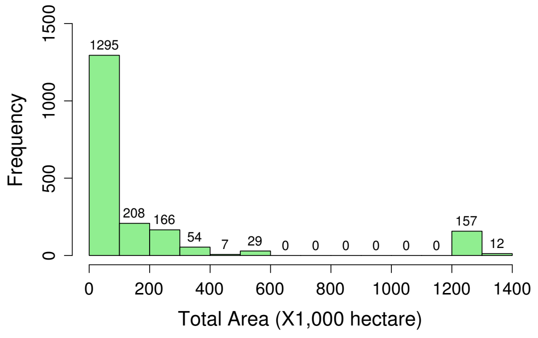

Figure 6: Histogram of total area in hectares per tile of Arealm (landmass polygons) table

the spatial join queries. For convenience, we upload the original dataset into PostgreSQL tables and issue SQL queries as part of the procedure call *retrieveOverlappingObjects()*. The actual work metric calculation (procedure *calculateMetric*) depends on which work metric to use. We evaluate three metrics in Section 4.3.

Once the declustering process completes, the generated tile list is iterated with Hilbert SFC order and for each object fragment a record is generated. The fields of the record include the tile id, original object id, the MBR of the object, the fragment id, the fragment geometry, the fragment MBR and other fields. These records are persisted in disk files to be later retrieved when the in-memory tables are populated. For parallel retrieval multiple disk files are used for a particular table.

When the query execution engine of SPINOJA starts, first each in-memory table is populated by reading the corresponding disk files. The schema of a given in-memory table *tab* is as follows:

{*ObjectId*, *ObjectMBR*, *FragmentId*, *FragmentMBR*, *FragmentGeom*, ... <other fields of *tab*>}

As part of the table loading process a tile bitmap index is also created. The index maps each fragment id to the tile id to which it belongs. The index is essentially a dictionary as shown below:

{**key**: *TileId*, **value**: compressed bitmap of *FragmentId*s}

The use of a compressed bitmap is meant to keep the memory footprint of the index low.

4.3 Work metrics

The declustering phase in SPINOJA attempts to create tiles with about the same amount of computation demands. Key to this process is choosing a good work metric – one that is able to estimate the actual amount of computation needed while processing a tile. We describe and compare two different work metrics: object size based and point density based.

4.3.1 Object size based metrics

A potential work metric candidate is the object size. Intuitively, the larger the object the more work is needed in the refinement step. Many spatial datasets, such as geographical and VLSI datasets, are characterized by significant variation in the sizes of the objects. To demonstrate, we calculated a frequency histogram of the total area of all the polygons in the tiles created by Clone Join declustering from the Arealm table. Figure 6 shows the histogram for tiles with non-zero area. As can be seen, in 169 out of 1928 tiles (8.7% of the tiles), the total area of all polygons is over 1200K hectares. Usually these tiles 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

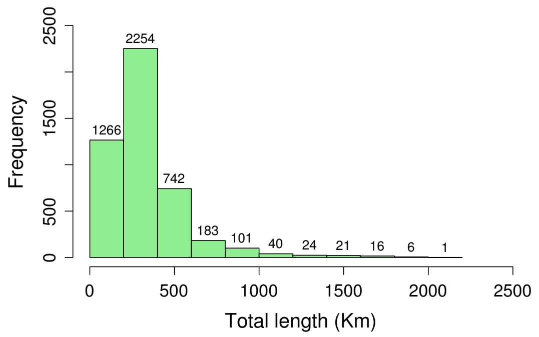

Figure 7: Histogram of total length in km per tile of Edges (polylines) table

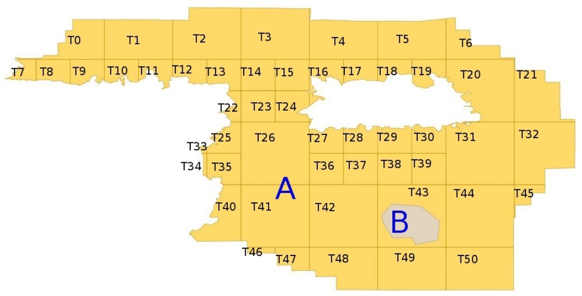

Figure 8: The cost of evaluating polygon overlaps polygon

other objects. In the rest (91.3%) of the tiles the total area of all polygons is less than 600K hectares. Note that the TIGER dataset already does a good job of refactoring very long polyline features (such as rivers, or roads) into smaller polyline segments. This is illustrated in Figure 7, which shows the frequency histogram of the total length for all polylines in each tile (with non-zero length) from the Edges table. To contrast the variability in object sizes between Arealm and Edges tables, we computed the standard deviation of the total area of all polygons and the total length of all polylines among the tiles in Arealm and Edges respectively. The standard deviation is 354335.2 for Arealm, whereas it is 235.5 for Edges. Therefore, it is expected that the processing skew would be less pronounced in “polyline and polyline” queries.

Since the MOD-Quadtree declustering decomposes the spatial objects, a metric based on object size would lead to creating smaller objects and hence reduce the overall computation. For a polygon the size of an object is its area, but for polyline objects the size is related to its length. To use a metric with uniform dimension we use the area of the MBR of an object or *AreaOfMBR* as the metric. Formally, the work incurred by tile t with this metric is given by,

$$W_t = \sum_{i=1}^{R} AreaOfMBR_i$$

where the number of objects in t is R.

4.3.2 Point density based metrics

The processing of a spatial predicate is dominated by the refinement step, which itself consists of two steps: an intersection determination step, based on the Bentley-Ottmann plane sweep algorithm [3], and a matrix creation step, based on the dimensionally extended 9-intersection model [6]. The dominant cost component of these two is the plane sweep algorithm, which has a running time of $O((n + k)\ \log n)$. Here n is the total number of points in all objects involved and k is the number of intersections in the output. Since k is not known in advance, this cost can be approximated by $O(n\ \log n)$ for relatively small k. In the existing approaches to spatial join, the actual geometry of each object from the first dataset must be compared with that from the second

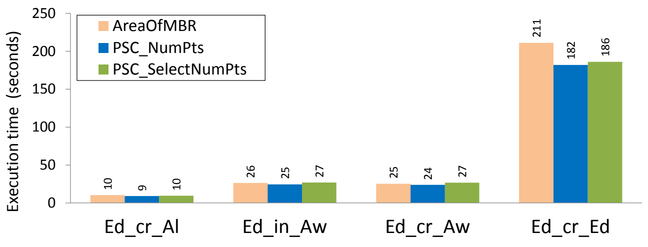

Figure 9: Execution times of queries with different metrics (2 cores)

dataset. For objects with many points this may incur significant processing cost in the plane sweep algorithm. For instance, in Figure 8 there are two polygons: A (representing San Bernardino Natl Forest with 11,931 points) from the first dataset and B (a small lake B with 9 points) from the second dataset. The cost of the plane sweep, to determine if B overlaps A, can be approximated as $(11931 + 9) * \log(11931 + 9) \approx 112078.5$.

In the SPINOJA approach, the objects are decomposed by clipping against the tile boundaries. Therefore, only the object fragments within each tile need to be compared. In Figure 8 there were 51 tiles generated by SPINOJA, T0 to T50. For demonstration purposes, we label the tiles sequentially from left to right and then top to bottom, rather than using Hilbert SFC order. As can be seen, object A is split into multiple polygons at the tile boundaries and object B is entirely contained by tile T43. Thus, only the object fragment of A contained within T43 needs to be evaluated. The cost of the plane sweep can be approximated as, $(4 + 9) * \log(4 + 9) \approx 33.3$, which is significantly lower than the original. As a result, we choose the plane sweep cost based on the number of points as another work metric. We call this *PSC_NumPts*. The work incurred by tile t with this metric is:

$$W_t = P_t * \log(P_t)$$

where P_t is the total number of points in all objects in tile t.

The factor k (the number of intersections found) in the true cost of Bentley-Ottmann’s plane sweep algorithm can be approximated by the selectivity of the spatial join predicate for a particular tile being evaluated. We use this as the third work metric and call it *PSC_SelectNumPts*. The work involved in tile t with this metric is:

$$W_t = Sel_t * R * P_t * \log(P_t)$$

where P_t is the total number of points in all objects in tile t, number of objects R and Sel_t the selectivity of t. For selectivity estimation any technique such as [1] can be used. Since Intersects is one of the least restrictive OGC predicate, we use the aggregate of the selectivity estimates for this predicate between each pair of tables.

4.3.3 Evaluation of the metrics

To evaluate the three metrics we executed the queries (Table 1) with 10K tiles generated by MOD-Quadtree declustering. We used the setup described in Section 5.1 with 2 cores. Figure 9 shows the execution times of 4 queries with the three partitioning metrics. We see that the PSC_NumPts metric performs best in all cases. Note that the difference in query execution times between the two point density metrics (PSC_NumPts and PSC_SelectNumPts) is not very significant. However, PSC_SelectNumPts requires the additional step of selectivity estimation and hence more computation.

Intuitively, a point density based metric may be superior since the cost of the plane sweep algorithm in the refinement step is dependent on the number of points. PSC_NumPts is a clear winner over AreaOfMBR, especially with the longest running query Ed_cr_Ed. Therefore, we use PSC_NumPts as the default metric in all subsequent experiments where applicable.

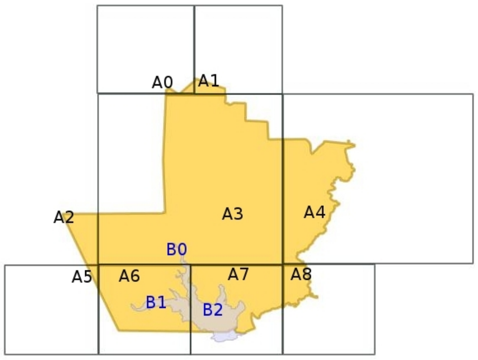

Figure 10: Processing of predicate Polygon overlaps polygon

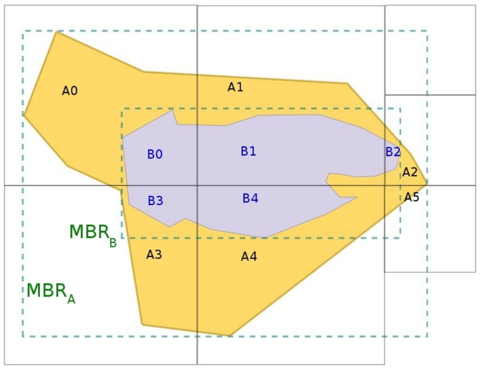

Figure 11: Processing of predicate Polygon contains polygon

4.4 Processing of spatial predicates

The spatial predicates describe how two spatial objects relate to each other in terms of topological constraints. A number of formal models have been been proposed, including the nine-intersection model [8], to characterize these topological relations. However, many of these relations are difficult to embed in a DBMS query language. Clementini and Di Felice proposed a model to derive a subset of the relationships in the nine-intersection model [6]. This model, known as the Dimensionally Extended Nine-Intersection Model (DE-9IM), has been adopted by OGC.

In the existing spatial join approaches that rely on spatial declustering, a *result aggregation* step is needed to remove duplicates from the resultset. This may be necessary because in a grid partitioning approach such as Clone Join, the same object can be replicated to multiple grid tiles. In SPINOJA, the objects are not replicated, but rather decomposed into fragments along tile boundaries. Therefore, the result aggregation step is different from other approaches and depends on the type of spatial predicate.

For some of the DE-9IM predicates, such as Intersects, Touches, Crosses and Overlaps, the satisfaction of the predicate by any one pair of the object fragments in one of the tiles is sufficient. This is illustrated in Figure 10. The polygon A has been decomposed into fragments A0 through A8. Another polygon B has been split into fragments B0, B1 and B2. Since the fragment pairs from each tile are evaluated individually, if the predicate is satisfied by *any one* fragment pair of A and B, the predicate is satisfied for the entire object. In this example, if the predicate is “A Overlaps B”, it is satisfied for fragment pairs A3 and B0, A6 and B1, and A7 and B2.

For several of the DE-9IM predicates, namely, Equals, Within and Contains, *all* the fragment pairs from every tile for object A and B must satisfy the predicate. In order to avoid costly refinement involving the fragments, after the filter step and prior to refinement we check to see if the MBRs of the *entire* objects satisfies the predicate, i.e., whether the predicate is satisfied by MBR_A and MBR_B. Whereas in the filter step the fragment MBRs are used, in this evaluation step the MBRs of the entire objects are used. This is illustrated in Figure 11. If the predicate is “A Contains B”, all the fragment pairs, namely, A0 and B0, A1 and B1, A2 and B2, A3 and B3, and A4 and B4 must satisfy the predicate Contains.

4.5 Load balancing

In a parallel query execution system load balancing is necessary to evenly distribute workload to worker threads to ensure that the overall query execution time is reduced. The aim of load balancing is to minimize the idle time by attempting to assign an equal share of work to the workers so that they are always kept busy. In practice, however it is not easy to allocate an equal amount of work to each worker. If a certain task takes much longer than others it will increase the idle times for the remaining worker threads and the slowest task will determine the overall query latency.

Load balancing could be based on static or dynamic work assignment. In the static approach, the workload is partitioned into equal sized tasks before assigning to the workers (e.g., Clone Join statically assigns the spatial partitions to workers in a round-robin fashion). This method cannot guarantee uniform idle times among the worker threads even when number of objects are the same per spatial partition. As noted in Section 3, some partitions may take much longer to process due to the properties of the objects. Dynamic work assignment addresses the load imbalance using a scheduler and a task queue. The scheduler enqueues the next task from a task pool until exhausted. Each worker dequeues an entry from the task queue and once the execution is complete it looks for the next task.

SPINOJA uses the dynamic task assignment approach. For each query job the scheduler creates many tasks, each with the id of a tile generated during the declustering step. These tasks are inserted into a synchronized task queue. Each worker thread picks the next task and performs the spatial join on the objects from both tables that belong to the corresponding tile. Since there are two distinct phases of filter and refinement, it is possible to use different strategies as to when to schedule the filter and refinement. We describe and evaluate three strategies.

In the first approach, the tuples from the left table belonging to the current tile are iterated over. For each such tuple, the tuples from the right table in the current tile are checked to see if their MBR satisfies the spatial predicate; if so, the actual geometries are compared immediately in the refinement step. This is essentially a per tile nested loop join and we call this approach *TileNLJ* (tile-wide nested loop join). The sketch of the algorithm is shown in Figure 12. The TaskManager creates and enqueues a task *SPJTask* for each tile (lines 1-3). A Worker retrieves the next task from the queue and obtains the tile bitmaps for the tileid from the left and right tile indexes (lines 10-12). Then for each entry in the left tile bitmap the fragment geometry is retrieved from *leftTable*. Similarly, in the inner loop the fragment geometries are retrieved from the *rightTable* (lines 13-16). In the filter step the MBRs of the two geometries are checked for intersection (line 18). If true, the refinement step is performed with the actual geometries and for the spatial predicate *spPredicate* (line 20). If the predicate is satisfied, the corresponding object ids are added to the local resultset (lines 21-23). When all the tasks are completed, the TaskManager retrieves the local resultsets and performs the result aggregation and duplicate elimination (line 5-7).

In the second approach, a tile-wide plane sweep is performed to do the filter and then the refinement step is immediately conducted on the candidate tuple pairs generated by the filter. The plane sweep technique, used in [4, 15, 17], involves sorting the object fragment MBRs on their lower x-coordinate. Then the MBR with the smallest x-value is chosen and all the MBRs from the other table that overlap it along the x-axis are chosen. Then those MBRs are checked for overlap along the y-axis, and candidate pairs are generated with the fragment ids of the matching MBRs. This process is repeated until all tuples from both the tables are merged. All the candidate pairs from a tile that pass the filter step then go through the refinement. We call this scheduling approach *TilePS* (tile-wide plane sweep) and the algorithm sketch is shown in Figure 13. The TaskManager steps for this are the same as in Figure 12. For a particular tile, the Worker retrieves and adds the MBRs of all fragment geometries from the *leftTable* and *rightTable* into two lists (lines 7-12). These two lists are used as inputs to the tile-wide plane sweep in the filter step (line 14). The fragment id pairs from two tables that satisfy this plane sweep step constitute the *candidate set*. The actual geometries for each member of this set are tested to see if they satisfy the predicate in the refinement step (lines 17-19).

The filter step of the third scheduling strategy is similar to that of the second approach. However, unlike an immediate refinement that follows the filter step, in this approach the refinement step is performed separately for a fixed size batch of candidate set. The idea behind this strategy is to evenly distribute the refinement load among the worker threads. Some tiles may produce more candidate set pairs in the filter step than other tiles. So if the refinement step is also performed by the same thread as the filter, some threads may end up doing more refinement work. For this reason, the candidate pairs generated in the filter step are grouped together in batches of size *BATCH* (we use *BATCH*=100) to create a new task. The Worker enqueues these tasks in the queue after the tile-wide plane sweep filter step. Since the refinement step is executed as part of a separate task, we call this scheduling approach *TilePS_sepRefine* (tile-wide plane sweep with separate refinement; Figure 14). Again, the TaskManager steps for this are the same as before. After the tile-wide plane sweep is performed (line 15), the Worker iterates over the candidate set and produces batches of size *BATCH*. For each batch it enqueues a task *SPJTask* of type *REFINE* (lines 17-23). When a Worker retrieves such a task from the queue, the actual geometries for each entry of the batch are obtained and the refinement step is performed (lines 26-30).

We implemented the three load balancing strategies and evaluate them with 10K tiles generated by MOD-Quadtree declustering. We assign each Worker to a different core using Linux setaffinity. We used the setup in Section 5.1 with 2 cores. Figure 15 shows the execution times of four queries with the load balancing approaches. TileNLJ does significantly worse than the other scheduling strategies with queries Ed_cr_Ed, but does slightly better with the other three queries. The candidate set size or the number of candidate pairs generated from the filter steps of these approaches with different queries are shown in Figure 16. Clearly, for Ed_cr_Ed query the candidate set size is significantly larger with TileNLJ than the other two strategies and this is reflected in the query execution times. Figure 17 shows the break-down of execution times of the scheduling strategies with two queries Ed_cr_Aw and Ed_cr_Ed. For scheduling approaches TilePS and TilePS_sepRefine there is a *Tile-wide plane sweep* step, whereas for TileNLJ there is a *Pairwise filter* step. The tile-wide plane sweep, as explained earlier, is a filter step performed for all the objects in a tile, whereas the pairwise filter is done for each pair of objects as part of the nested loop in TileNLJ. The tile-wide plane sweep does a better job of reducing the candidate set size than the pairwise filter (as is evident from Figure 16), but takes longer. The overhead of this step causes the queries to take longer with TilePS and TilePS_sepRefine than TileNLJ for the polyline and polygon queries. However, for Ed_cr_Ed query this overhead is more than compensated by the reduction of time in refinement due to the much smaller candidate set, as can be seen in Figure 17, This suggests that for queries with relatively smaller filter selectivity, TileNLJ is the best approach, but for queries with a large candidate set TilePS is a better option. It is possible to design a *hybrid* strategy that chooses either TileNLJ or TilePS depending on the filter selectivity.

Interestingly, TilePS_sepRefine does slightly worse than TilePS. To see why, we present the last level cache misses with TilePS and TilePS_sepRefine in Figure 18. As can be seen, TilePS exhibits fewer cache misses for all queries. With this strategy, the same worker thread performs both the filter and refinement step for the objects of a particular tile and so the queries with TilePS have better data locality. We use TilePS as the default load-balancing strategy unless otherwise specified.

```
Require: leftTable and rightTable are the 2 tables, and leftTileIndex and
    rightTileIndex are the corresponding indexes; spPredicate is the spa-
    tial join predicate
TaskManager:
 1: while tileId in leftTileIndex do
 2:     Create a new SPJTask with tileId
 3:     TaskQueue.push(SPJTask)
 4: {//Wait for all tasks to complete}
 5: for worker in listOfWorkers do
 6:     localResultset ← worker.getLocalResultset()
 7:     update query resultset with localResultset
Worker:
 8: while true do
 9:     SPJTask ← TaskQueue.pop()
10:     tileId ← SPJTask.tileId()
11:     leftObjBitmap ← leftTileIndex.getBitmap()
12:     rightObjBitmap ← rightTileIndex.getBitmap()
13:     for leftBitIdx in leftObjBitmap do
14:         leftGeom ← leftTable.getGeomAtFragId(leftBitIdx)
15:         for rightBitIdx in rightObjBitmap do
16:             rightGeom ← rightTable.getGeomAtFragId(rightBitIdx)
17:             {//Filter step: pairwise MBR compare}
18:             if MBRIntersects(leftGeom.MBR,rightGeom.MBR) then
19:                 {//Refinement step}
20:                 if satisfies(spPredicate,leftGeom,rightGeom) then
21:                     leftObjId ← leftTable.getObjId(leftBitIdx)
22:                     rightObjId ← rightTable.getObjId(rightBitIdx)
23:                     UpdateLocalResultset(leftObjId,rightObjId)
```

Figure 12: Algorithm Load-balance TileNLJ

```
TaskManager:
 1: {//same as before}
Worker:
 2: while true do
 3:     SPJTask ← TaskQueue.pop()
 4:     tileId ← SPJTask.tileId()
 5:     leftObjBitmap ← leftTileIndex.getBitmap()
 6:     rightObjBitmap ← rightTileIndex.getBitmap()
 7:     for leftBitIdx in leftObjBitmap do
 8:         leftGeom ← leftTable.getGeomAtFragId(leftBitIdx)
 9:         leftMBRList.add(leftGeom.MBR,leftBitIdx)
10:     for rightBitIdx in rightObjBitmap do
11:         rightGeom ← rightTable.getGeomAtFragId(rightBitIdx)
12:         rightMBRList.add(rightGeom.MBR,rightBitIdx)
13:     {//Filter step: tile-wide plane sweep}
14:     rsFragmentIdPairList ← PlaneSweep(leftMBRList,rightMBRList)
15:     {//Refinement step}
16:     for fragIdPair in rsFragmentIdPairList do
17:         leftGeom ← leftTable.getGeomAtFragId(fragIdPair.leftId)
18:         rightGeom ← rightTable.getGeomAtFragId(fragIdPair.rightId)
19:         if satisfies(spPredicate,leftGeom,rightGeom) then
20:             leftObjId ← leftTable.getObjId(fragIdPair.leftId)
21:             rightObjId ← rightTable.getObjId(fragIdPair.rightId)
22:             UpdateLocalResultset(leftObjId,rightObjId)
```

Figure 13: Algorithm Load-balance TilePS

```
TaskManager:
 1: {//same as before; Create tasks SPJTask with type FILTER}
Worker:
 2: while true do
 3:    SPJTask ← TaskQueue.pop()
 4:    if SPJTask.getType = FILTER then
 5:       tileId ← SPJTask.tileId()
 6:       leftObjBitmap ← leftTileIndex.getBitmap()
 7:       rightObjBitmap ← rightTileIndex.getBitmap()
 8:       for leftBitIdx in leftObjBitmap do
 9:          leftGeom ← leftTable.getGeomAtFragId(leftBitIdx)
10:          leftMBRList.add(leftGeom.MBR,leftBitIdx)
11:       for rightBitIdx in rightObjBitmap do
12:          rightGeom ← rightTable.getGeomAtFragId(rightBitIdx)
13:          rightMBRList.add(rightGeom.MBR,rightBitIdx)
14:       {//Filter step: tile-wide plane sweep}
15:       rsFragIdPairList ← PlaneSweep(leftMBRList,rightMBRList)
16:       {//Enqueue Refinement task}
17:       Create a new SPJTask with type REFINE
18:       for fragIdPair in rsFragIdPairList do
19:          SPJTask.add(fragIdPair)
20:          if SPJTask.batchSize() = BATCH then
21:             TaskQueue.push(SPJTask)
22:             Create a new SPJTask with type REFINE
23:       TaskQueue.push(SPJTask)
24:    else
25:       rsFragmentIdPairList ← SPJTask.getFragmentIdPairList()
26:       for frIdPair in rsFragmentIdPairList do
27:          leftGeom ← leftTable.getGeomAtFragId(frIdPair.leftId)
28:          rightGeom ← rightTable.getGeomAtFragId(frIdPair.rightId)
29:          {//Refinement step}
30:          if satisfies(spPredicate,leftGeom,rightGeom) then
31:             leftObjId ← leftTable.getObjId(frIdPair.leftId)
32:             rightObjId ← rightTable.getObjId(fragIdPair.rightId)
33:             UpdateLocalRefineResultset(leftObjId,rightObjId)
```

Figure 14: Algorithm Load-balance TilePS_sepRefine

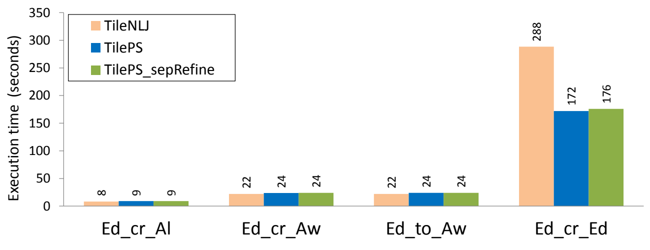

Figure 15: Execution times of queries with different load-balancing strategies (2 cores)

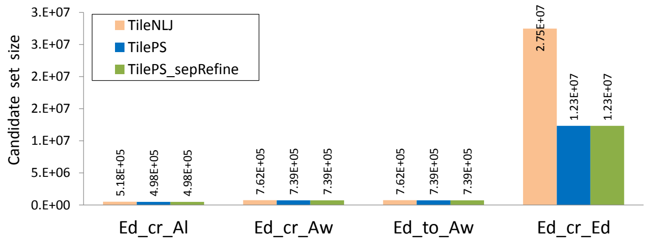

Figure 16: Candidate set size of queries with different load-balancing strategies (2 cores)

4.6 Determining the number of partitions (tiles)

An important question in spatial declustering is how many tiles to create in order to achieve the best query execution time. In existing declustering approaches that replicate objects to all tiles that they overlap, there is a trade-off. The more tiles that are created, the more object replication is needed because of the increased probability of overlapping with the tile boundaries, leading to more memory consumption. However, more tiles imply smaller tiles with fewer objects and faster processing time per tile.

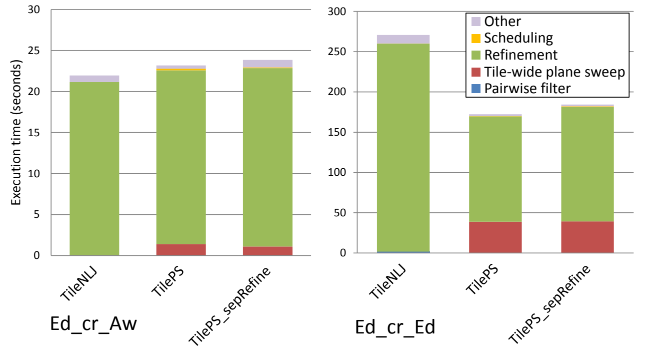

Figure 17: Break-down of time with different load-balancing strategies (2 cores)

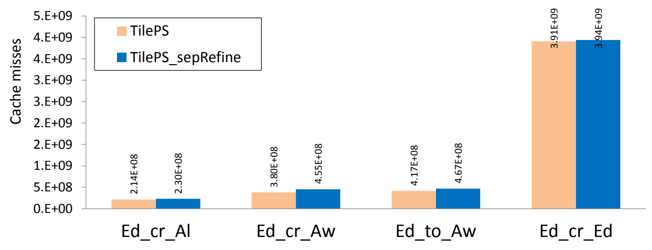

Figure 18: Cache misses with different load-balancing strategies (2 cores)

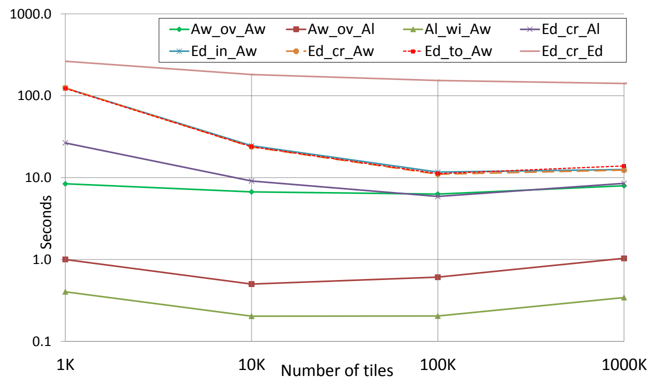

Figure 19: Execution times of queries with different number of tiles (2 cores)

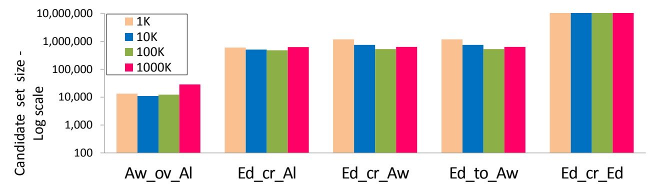

Figure 20: Candidate set size of queries with different number of tiles (2 cores)

Since SPINOJA's MOD-Quadtree declustering does not replicate objects, but rather decomposes objects along tile boundaries, the increase in memory usage is less significant. Although smaller tiles incur less processing costs, if there are too many tiles the scheduling overhead may outweigh the benefits of partitioning. To explore these issues, we evaluate SPINOJA with 1K, 10K, 100K and 1000K tiles using the setup in Section 5.1 with 2 cores. Figure 19 shows the query execution times with the four tile number configurations.

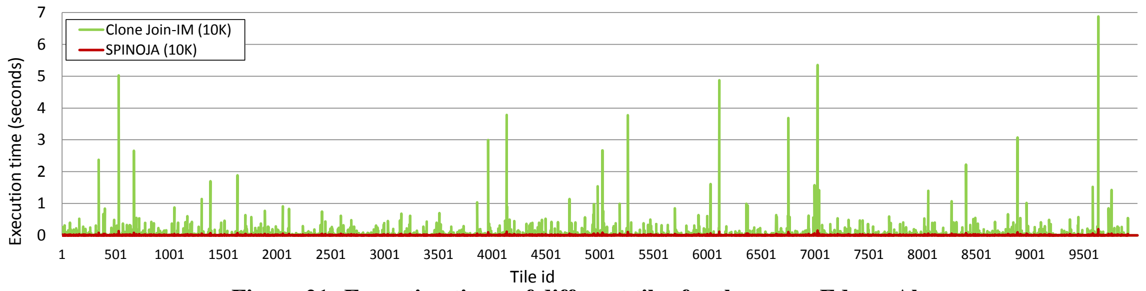

Figure 21: Execution times of different tiles for the query Ed_cr_Al

Table 2: Point density per tile for Ed_cr_Ed

1K	10K	100K	1000K
43494.7	4329.4	444.2	50.75

As expected, 1K tiles results in the worst execution times for all queries. For short-running queries (i.e., overall latency of less than a second) the best performance is achieved with 10K tiles. From 1K to 10K tiles, the execution times of all queries improve. For relatively longer-running queries (i.e., overall latency greater than 10 seconds) the performance further improves when the number of tiles is increased from 10K to 100K. Beyond 100K tiles, the performance of all queries worsens, except for the longest-running Ed_cr_Ed. To explain the result, in Figure 20 we plot the candidate set size generated by the filter step of some of the queries for different numbers of tiles. As can be seen, for the short-running query Aw_ov_Al, the candidate set size is the smallest for 10K tiles. For the longer running queries, Ed_cr_Al, Ed_cr_Aw and Ed_to_Aw, it is the smallest for 100K tiles. The candidate set size remains relatively unchanged for the query Ed_cr_Ed. However, since the point density per tile decreases (Table 2) as the number of tiles increases, the amount of work per tile is reduced. Therefore, the execution time for Ed_cr_Ed continues to decrease as the tile count increases.

4.7 Managing processing skew

Parallel spatial join query execution is susceptible to processing skew due to the dataset properties. Object replication based declustering may aggravate the situation. As shown in Figure 2, although the vast majority of tiles take less than 0.5 seconds to process the Ed_cr_Al query with in-memory Clone Join (which we call Clone Join-IM), a few take significantly longer. SPINOJA was designed to address this processing skew. To show how well it achieves this goal, we execute the Ed_cr_Al query with 10K tiles and plot the execution times of each individual tile in Figure 21. As can be seen, none of the tiles takes more than 0.2 seconds. To contrast, we also plot the execution times of the tiles with Clone Join-IM. The standard deviation of the tile execution times is 0.167 for Clone Join-IM and 0.006 for SPINOJA. Thus, it is apparent that SPINOJA does a much better job in reducing the processing skew and hence improving on the overall query execution time.

5. EXPERIMENTAL EVALUATION

In this section we evaluate SPINOJA in various settings. We first describe the datasets and the settings used in our experiments.

5.1 Experimental setup

We use a real-world spatial dataset that contains diverse geographical features, drawn from the TIGER® data [20], produced by the United States (US) Census Bureau. This is a public domain data source available for each US state. The dataset that we use consists of the polyline and polygon shapefiles for all the counties of California. We merged the shapefiles to create single shapefiles.

Table 3: Database tables

Dataset	Database table	Geometry	Cardinality
California	Edges	polyline	4173498
	Arealm	polygon	7953
	Areawater	polygon	39334

Table 3 shows the details of each, including the the geometry and cardinality. We use the best of 3 warm runs to report the query execution time results for the queries in Table 1.

The experiments were conducted on a machine having 16 GB memory, eight 3.0 GHz Intel Xeon CPU cores with 6 MB cache, and an 880 GB 7200-RPM SATA disk. We run Ubuntu 10.04 Lucid 64-bit with kernel version 2.6.32-33-generic as the OS.

5.2 Multicore scalability

SPINOJA's declustering mechanism already does a good job of reducing the overall query execution time by minimizing the processing skew. Since it is a parallel spatial join approach, we are also interested in its multicore scalability. To evaluate the multicore performance we executed the queries with 2, 4, 6 and 8 cores. We use TilePS as the load-balancing strategy and 100K tiles.

Figure 22 shows the speedup achieved with 2, 4, 6 and 8 cores over the the execution times with 1 core. All queries with latency more than 1 second exhibit significant decrease in execution time, obtaining benefit from the addition of cores. These queries show a "staircase" pattern in the reduction in latency. With SPINOJA the queries achieve near linear speedup in the number of cores.

5.3 Comparison with other in-memory spatial join approaches

To evaluate the performance of SPINOJA against in-memory spatial join approaches we implemented TOUCH [12]. We also implemented **in-memory** versions of Clone Join [16], INLJ (indexed nested loop join) and NLJ (nested loop join). Both TOUCH and INLJ use a bulk-loading in-memory R-tree variation called STR-tree that offers the best query performance. Since SPINOJA is a parallel spatial join, in order to do a fair comparison we **parallelized** each of these approaches. The Join Phase of TOUCH was parallelized by assigning cells to different threads in a round-robin manner. We call this parallel version **TOUCH-P**. The other phases of TOUCH (Tree Building and Assignment) were processed using pre-processing steps and are not part of the comparison.

The INLJ approach was parallelized by range partitioning the larger table among the worker threads and creating an STR-tree index on the other table. Similarly NLJ was parallelized by range partitioning the larger table among different threads. We found that the NLJ approach takes too long for queries with Edges and Areawater tables and so we omit NLJ from the results.

First, we report the memory usage of the different approaches. With Clone Join the number of tiles is a crucial factor in the overall memory consumption. The authors of Clone Join used 10K tiles in the spatial declustering function for their experiments and

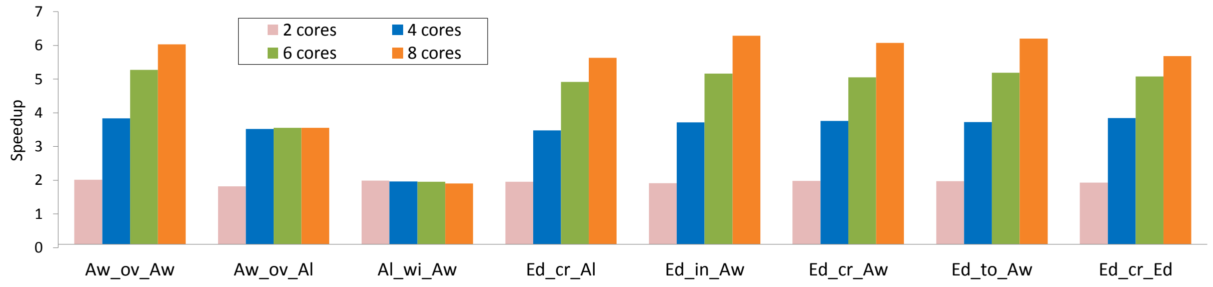

Figure 22: Multicore speedup of SPINOJA over 1 core performance (2, 4, 6 and 8 cores)

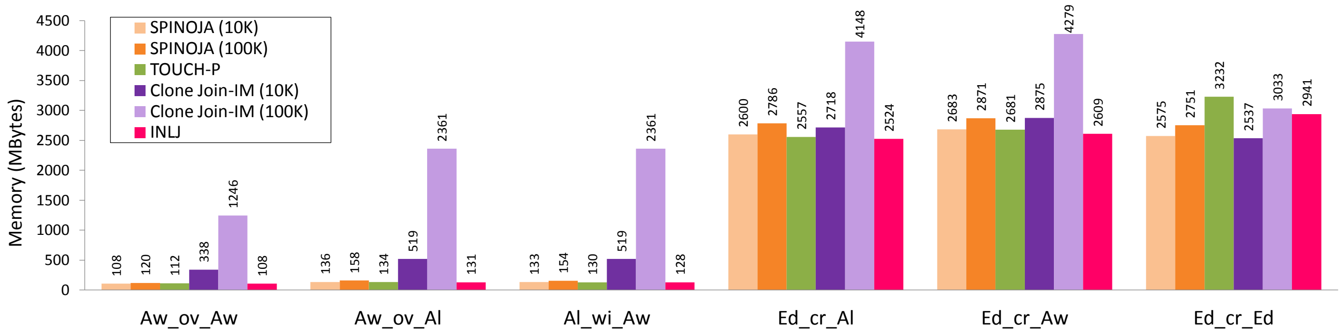

Figure 23: Memory usage for queries with SPINOJA vs. other approaches (8 cores)

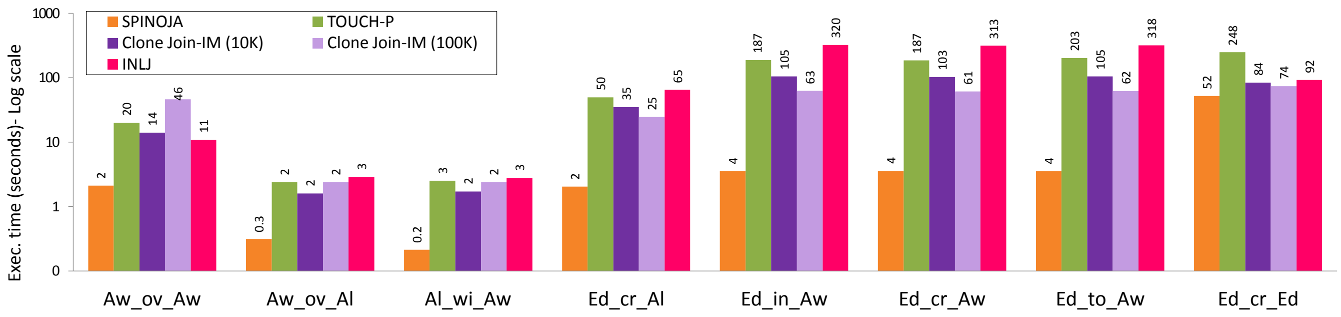

Figure 24: Execution times of queries with SPINOJA vs. other approaches (8 cores)

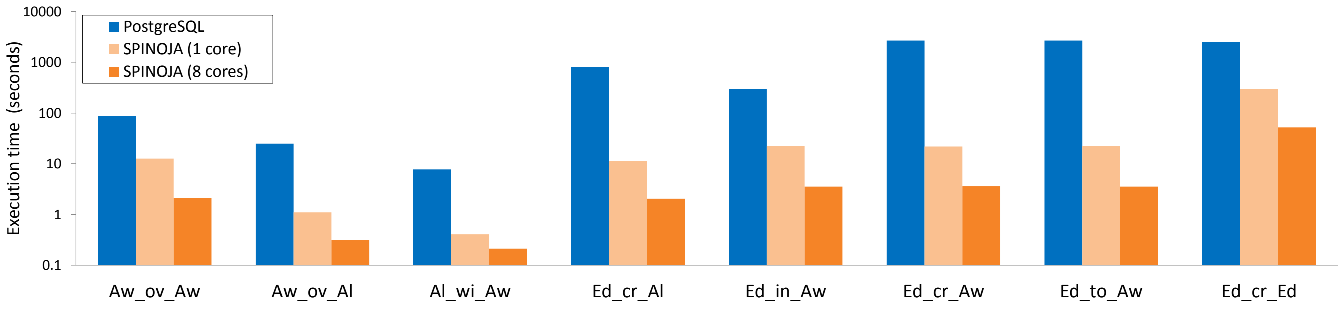

Figure 25: Execution times of queries with SPINOJA vs. PostgreSQL

did not report the effect of the number of tiles. We implemented in-memory versions of Clone Join with both 10K and 100K tiles and call them **Clone Join-IM (10K)** and **Clone Join-IM (100K)** respectively. To compare, we also present the memory usage of SPINOJA with the same tile configurations. As shown in Figure 23, Clone Join-IM with 100K tiles requires significantly more memory than all other approaches. This is because with more tiles the probability of an object overlapping multiple tiles increases and hence the likelihood of it getting replicated to each such tile also increases. SPINOJA uses slightly more memory with 100K tiles than with 10K, but is still comparable to the memory usage of TOUCH-P.

For SPINOJA we report the execution times with 100K tiles, as this gives the best performance for the longer running queries. The execution times of the queries with SPINOJA and the other approaches are shown in Figure 24. All the query execution times are reported for the 8-core execution. It is evident that SPINOJA outperforms all other approaches by wide margins. SPINOJA is 90X faster than INLJ in the best case. Compared to TOUCH-P, SPINOJA is significantly faster, particularly, with the queries involving Edges and Areawater. For instance, SPINOJA is about 50X faster than TOUCH-P with Ed_cr_Aw query. Note that TOUCH-P does better than INLJ in most queries but TOUCH-P performs worse with self-join, namely Aw_ov_Aw and Ed_cr_Ed. This is because its Assignment step is unable to filter objects for self-join.

Between the two Clone Join variations, there is no clear winner in terms of execution time: Clone Join-IM (10K) does better than Clone Join-IM (100K) with the short running queries and Clone Join-IM (100K) performs better than the other with the longer running queries. In fact, Clone Join-IM (100K) is faster than both TOUCH-P and INLJ with the longer running queries. However, Clone Join-IM (100K) requires significantly more memory than Clone Join-IM (10K). Due to this increased memory usage, Clone Join-IM (100K) may not be practical with larger datasets. SPINOJA does better than Clone Join-IM (100K) and Clone Join-IM (10K) in all cases. For instance, with Ed_to_Aw query SPINOJA is about 30X faster than Clone Join-IM (10K) and 17X faster over Clone Join-IM (100K). With the purely polyline based query Ed_cr_Ed, SPINOJA achieves the least speedup over the other approaches because, this query exhibits less processing skew than the others, as explained in Section 4.3.1.

5.4 Comparison with PostgreSQL

SPINOJA is essentially a main-memory column store database optimized for spatial join. A direct comparison with a general purpose relational database, such as PostgreSQL, cannot be made because SPINOJA lacks several features such as transactions and a query optimizer. However, by comparing SPINOJA with PostgreSQL we make the case that a custom-built database might be better suited for spatial join than a general purpose relational database. In Figure 25 we compare the execution times of the queries with PostgreSQL 8.4.4. The bufferpool of PostgreSQL was set to 8 GB so that the database completely fits in memory. Since PostgreSQL does not yet support intra-query parallelism, we present the query execution times with SPINOJA for one core, alongside those with 8 cores. The single-core performance of SPINOJA over PostgreSQL is quite good for both short and long-running queries. For instance, SPINOJA attains a speedup of 19X with Al_wi_Aw and a speedup of 123X with Ed_cr_Aw query. Consequently, the 8-core speedup of SPINOJA over PostgreSQL is significant. For example, this speedup is 749X with Ed_cr_Aw. Note that with the Intersects predicate (i.e. Ed_in_Aw query), SPINOJA achieves relatively less speedup than other "polyline and polygon" queries with different predicates. This is because PostgreSQL uses an optimization with this predicate, which SPINOJA currently does not implement.

6. CONCLUSIONS

Spatial join is important in many traditional and emerging spatial data analysis applications. We have introduced SPINOJA, a parallel in-memory spatial query execution infrastructure. It is designed to accelerate spatial join performance by exploiting large main-memory available in modern machines and rising core count.

Previous approaches to disk based parallel spatial join used spatial declustering that attempted to distribute the objects to tiles based on object count. Moreover, they focused on the filter step. However, when the more compute-intensive refinement step is taken into account, the object properties, such as size or point density, become important. As a result, even when the number of objects per tile is roughly equal, processing skew can occur. In an in-memory spatial join query, in the absence of disk latency, the processing skew becomes the key bottleneck to parallel performance.

SPINOJA addresses this processing skew by using an object decomposition based declustering. The declustering uses a work metric to equalize the amount of computation demanded by each tile. We also present three load balancing strategies. With extensive evaluation we demonstrate that SPINOJA does significantly better than in-memory implementations of previous parallel spatial join approaches and the parallel performance of a recently proposed in-memory spatial join.

References

[1] A. Aboulnaga and J. F. Naughton. Accurate Estimation of the Cost of Spatial Selections. In *ICDE*, pages 123–134, 2000.

[2] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System over Mapreduce. *PVLDB*, pages 1009–1020, 2013.

[3] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, 28(9):643–647, 1979.

[4] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD*, pages 237–246, 1993.

[5] T. Brinkhoff, H. peter Kriegel, and B. Seeger. Parallel Processing of Spatial Joins Using R-trees. In *ICDE*, pages 258–265, 1996.

[6] E. Clementini and P. Di Felice. A model for representing topological relationships between complex geometric features in spatial databases. *Information Sciences*, pages 121–136, 1996.

[7] A. Colantonio and R. Di Pietro. CONCISE: COmpressed 'N' Composable Integer SEt. *Information Processing Letters*, 110:644–650, 2010.

[8] M. J. Egenhofer and R. Franzosa. Point-set topological spatial relations. *Intl Journal of GIS*, pages 161–174, 1991.

[9] R. H. Güting. An introduction to spatial database systems. In *VLDB*, pages 357–399, 1994.

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

[11] N. Koudas and K. C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.

[12] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki. Touch: In-memory spatial join by hierarchical data-oriented partitioning. In *SIGMOD*, pages 701–712, 2013.

[13] Open Geospatial Consortium. Simple Feature Access - Part 2: SQL Option. http://www.opengeospatial.org/standards/sfs.

[14] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *SIGMOD*, pages 326–336, 1986.

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

[16] J. M. Patel and D. J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *SIGSPATIAL*, pages 54–61, 2000.

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

[18] S. Ray, B. Simion, and A. D. Brown. Jackpine: A benchmark to evaluate spatial database performance. In *ICDE*, pages 1139–1150, 2011.

[19] S. Ray, B. Simion, A. D. Brown, and R. Johnson. A Parallel Spatial Data Analysis Infrastructure for the Cloud. In *SIGSPATIAL*, pages 274–283, 2013.

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

[21] 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.

[22] X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. *Geoinformatica*, pages 175–204, 1998.