# Response to the Reviewers Comments

On Scalable DCEL Overlay Operations

We would like to thank the reviewers for their valuable comments and constructive feedback. We have considered all their comments to revise the manuscript as detailed below in the responses to each comment. The corresponding changes in the revised manuscript are highlighted with **boldface blue font** for the convenience of the reviewers and editors.

## Reviewer 1

**The paper implements overlay operators using the Apache Spark framework, however, a map-reduce description of input partitioning, load balancing, local operations, and optimizations, with Spark-specific transformations and actions for the implementation, is not presented in any level of detail.**

We aimed to minimize platform-specific technical references. However, in the revised manuscript, we included additional details about Apache Spark's specific transformations and actions. Notably, we have incorporated the following content in Section 7:

***"The scalable approach was implemented using the Apache Spark framework. From a MapReduce perspective, the stages outlined in Section 4 were executed using various Spark-supported transformations and actions. For instance, the partitioning and load balancing methods described in Sections 4.1 and 6.3 were achieved using a Quadtree structure, where its leaves were utilized to map and balance the number of edges distributed to worker nodes. Primarily, map operations were employed to process and assign edges to the corresponding leaves, leveraging proximity while distributing workload across worker nodes. Similarly, edges within each partition were processed through chains of local transformations (see Section 4), followed by reducer actions to manage incomplete faces that spanned multiple partitions. These faces had to be merged or redistributed to produce the final output. Moreover, the reduce actions were further optimized as described in Section 5."***

**The experimental section includes new content, different from the published conference papers, however, the impact of shuffle, i.e., communication cost warrants more discussion.**

Thank you for your comment. We assume it pertains to the data shuffling introduced in the new experiments, specifically those related to Kdtree partitioning and the handling of polygons with dangle and cut edges. The shuffling process for the Kdtree partitioning is similar to that used in Quadtree partitioning, where all edges are shuffled. However, this shuffling only occurs once to group edges spatially based on proximity, ensuring they are clustered accordingly. After this initial step, the DCEL operations proceed with minimal data movement. For instance, during the overlay optimization, data shuffling is confined to the edges of incomplete faces that span multiple cells, which are typically few and relatively small. Importantly, the edges of these incomplete faces are distributed based on labels (using the *By Label* method), enabling parallelization of the shuffling phase and further reducing its computational impact. We have added **Table 5** and additional explanations in Section 7.1 of the paper to clarify this. A similar strategy is applied to handle dangle edges. Here, data shuffling is restricted to polygons and their corresponding dangle edges that span multiple cells. Given that such polygons are relatively few, the overall impact of this shuffling remains small. As with incomplete face edges, this shuffling is also label-based and can be parallelized.

**Similar to the original conference papers, sections 1-3 are well-written and easy to follow. Parenthesized statements throughout the paper can be stated directly if they are relevant to the text/discussion/experiment.**

Thank you for bringing this to our attention. We have corrected the parenthetical statements throughout the manuscript.

**Abstract:**
**0. Please clarify if the approach is only applicable to planar/projected data. In the case of data with only a Geographic Coordinate System, why would the current approach not be applicable?**

Our approach is applicable not only to planar or projected data but also to geographic data. In the revised manuscript, we have added the following content to the abstract.

***"The Doubly Connected Edge List (DCEL) is an edge-list structure widely used in spatial applications, primarily for planar topological and geometric computations. However, it is also applicable to various types of data, including 3D models and geographic data."***

**Section 4:**
**1. Could you provide a reference to the sequential base implementation used to test DCEL generation on the Main US dataset?**

In the revised manuscript, we included a reference to the baseline approach. The following content has been added to Section 7.

***"To test the scalable approach, a sequential algorithm for DCEL creation was implemented based on the pseudo-code outlined in [6]."***

**Section 4.1:**
**2. It is not clear what the authors mean by "(in the number of edges)".**

When we refer to '(in the number of edges)' we are highlighting the cells that contain considerably more edges in one layer compared to the other. In the revised manuscript, we have clarified this point by updating the text. The following content has been added to Section 4.1:

***"While a simple grid could be used to divide the spatial area, our early experiments demonstrated that this approach leads to unbalanced cells, with some containing significantly more edges than others, negatively impacting overall performance."***

**Section 4.2:**
**3. Can the authors provide more insight on the average and worst cases on how long it takes for Algorithm 1 to converge on a miss, i.e., when all three cells are orphans?**

In the revised manuscript, we included a running time analysis. The worst case would be the longest path of a quadtree which could be $O(N)$. However, in the average case, when the quadtree is balanced, the complexity would be logarithmic, $O(\log(N))$. The following content has been added to Section 4.2:

***"To determine the worst-case performance of the search algorithm, consider that for an orphan cell, the algorithm performs three point quadtree queries to find the sibling leaves containing the centroid. It then selects one of these leaves and repeats the process, querying three points for a new centroid within the siblings of the selected leaf. This causes the algorithm to explore progressively deeper into the quadtree. In the worst case, the longest path in the quadtree could result in a time complexity of $O(N)$. However, in the average case, when the quadtree is balanced, the complexity is logarithmic."***

**Section 5:**
**4. "boundaries of faces that expand different cells". expand vs span - is a cell expanding vs is a face extended/spanning over multiple cells? Currently, it makes the reader believe the cell is being expanded during a quadtree build, which in reality isn't the case. Please clarify the text and Section 5.1 title.**

We have updated the title of section 5.1 to ***"Optimizations for faces spanning multiple cells"*** and further clarified this concept throughout the section.

**Section 5.1:**
**5. "the user can specify a level in the quadtree structure (measured as the depth from the root) that can be used to combine cells together". In reality, a user has no prior insight for how the quadtree will look as it will depend on the data. Even worse, the user cannot predict the impact of selecting a level for reduction. The effect of data partitioning, and shuffle time has an impact on achievable parallelism. Section 5 and Section 7 do not explain these effects when considered together in the application.**

Although determining an intermediate-level value in advance can be challenging, it can be estimated based on the size of the input and the number of partitions, which the user knows. We did not explore this point further, as Section 7.1 presents alternatives with better performance. The following content has been added to Section 5.1:

***"While it may be challenging to predetermine an optimal level, it can be estimated based on the input size or the number of partitions. Moreover, Section 7.1 offers recommendations for suitable values and alternative approaches."***

**6. Overall, this section is difficult to follow without better presenting operations from a map-reduce standpoint. At each stage, what happens at the executors versus what happens at the driver? When do the jobs wait to synchronize or communicate? From a distributed computing standpoint, this information is not available to a reader.**

In the revised manuscript, we provide a detailed explanation of our approach from a Map-Reduce perspective. The following content has been added to Section 5.1:

***"From a distributed perspective, this process follows a typical Map-Reduce pattern. In the map phase, each worker node identifies and reports faces that are fully contained within its boundaries, as well as segments of faces that may need to be concatenated with segments reported by other nodes. These face segments are then sent to a master node, incurring communication costs as the master must wait for all nodes to report their segments. In the reduce phase, the master node groups the segments by face ID, sorts them, and concatenates the parts to form complete, closed faces."***

***"From a Map-Reduce standpoint, this alternative functions similarly to the previous approach but introduces additional reduce operations at an intermediate level. However, this also introduces new synchronization points, as each intermediate reducer must wait for its workers to report potential face segments before processing them. The reducer then either reports completed faces or sends incomplete segments to the driver for further processing."***

***"From a distributed computing perspective, this alternative introduces a shuffle stage at the beginning, eliminating the need for a reduce operation. The shuffle ensures that all segments with the same face ID are placed in the same worker, allowing them to be processed and reported directly."***

**Section 5.2:**
**7. "In the current approach" - did you mean in the naive approach?**

We refer to our initial implementation and have revised the statement in Section 5.2 for clarity.

***"In our initial implementation, the input sets of half-edges within each cell were combined into a single dataset, initially ordered by the x-origin of each half-edge."***

**Section 6.4:**

**8. The operations described in this section would be easier to follow with an illustration using the running example in the paper.**

In the revised manuscript, we have included new illustrations, ***Figures 14-16***, to better demonstrate the approach used. The following content has been added to Section 6.4:

***"Figure 14 illustrates the spatial partitioning of the two input layers, $A$ and $B$. Layer $A$ contains two input polygons, $A_0$ and $A_1$, while Layer $B$ consists of three dangle edges, $B_0$, $B_1$, and $B_2$."***

***"In Figure 15, Polygon $A_0$ is re-partitioned along with the edges it intersects, specifically $B_0$, $B_1$, and $B_2$."***

***"Figure 16 shows the result of polygonizing the edges from Polygon $A_0$ and $B_0$, $B_1$, and $B_2$, resulting in two polygons, $A_01$ and $A_02$."***

**Section 7.1:**

**9. The section will benefit from stats such as the number of holes, empty cells, and orphan cells generated.**

We have added a new ***Table 7***, which provides statistics on the number of cells, number of holes, and number of orphans (both cells and holes), highlighting the characteristics of the datasets. The following content has been added to Section 7.3:

***"Table 7 presents the number of cells, original holes, and the orphan cells and holes generated after partitioning."***

**10. "for each state experiment, we tried different numbers of leaf cells for the quadtree and presented the one with the best performance.". Please state the quadtree parameters to help the reader make an informed selection decision.**

In the revised manuscript, we provided details about the parameters used to construct the quadtrees for each state. We sampled 1% of the edges and evaluated the number of cells, varying from 200 to 2000. Through this analysis, we found that the best performance was achieved with approximately 3000 cells. The following content has been added to Section 7.1:

***"Note that for each state experiment, we tested different numbers of cells for the quadtree and reported the configuration with the best performance. To determine this, we sampled 1% of the edges for each state and evaluated the best number of cells ranging from 200 to 2000. In most cases, the best number of cells was around 3000."***

**Section 7.2:**

**11. The experiments demonstrate a speedup of the order of milliseconds. The contribution of the filtered-sweep algorithm in the final runtime on large datasets is not available.**

The performance of the filtered-sweep algorithm is significantly impacted by the number of edges in the input layers for each node. It is particularly effective when one set is much larger than the other. However, our real datasets do not exhibit this characteristic. As shown in Table 4, the total number of edges between layers is relatively similar, and this holds true for most cells as well.
As a result, we generated a synthetic dataset by starting with two real layers and adding more edges to one of them (Figure 18.a). We then selected the most diverse nodes and applied this optimization to those nodes (Figure 18.b). Since the layers in our real datasets have a similar number of edges, we do not expect significant performance improvements from this optimization. However, it could be beneficial for datasets where the two layers have drastically different edge counts. We have included the following clarification in the paper.

***"We anticipate that this optimization will be particularly beneficial for datasets where the two input layers contain many cells with significantly different edge counts."***

**Section 7.3:**
**12. The parenthesized statements can be stated directly if they are relevant to the experiment.**

Thank you for your feedback. We have revised the parenthetical statements in Section 7.3 and throughout the manuscript.

**13. It is not clear what impact data shuffle has on the runtime, if any. Specifically, does the size of data partitions have an impact on the runtime? What is the min/max/average size of a cell?**

We agree that the size of data partitions affects overall performance. To clarify this, we added **Table 6**, which provides statistics on the size of each cell. In larger datasets, we observed that an average cell size of approximately 3000 edges yields the best results. This cell size results in a relatively small amount of data to transmit, minimizing the impact on data shuffling and processing. The following content has been added to Section 7.3:

***"Additionally, Table 6 provides statistics on the cells. It shows that in larger datasets, an average cell size of approximately 3000 edges produces the best results. This cell size ensures a relatively small amount of data to transmit, which minimizes the impact on data shuffling and processing."***

**14. It is not clear how the quadtree splitting criteria is being controlled to create cells as a multiple of 1000 and not a multiple of 4.**

The splitting criteria in a quadtree is determined by the *maximum capacity* parameter. In the revised manuscript, we explained the key details of the quadtree splitting process and clarified why the final number of leaves is not always a multiple of four. The following content has been added to Section 7.3:

***"The quadtree configuration allows for performance tuning by setting the maximum capacity of a cell. The quadtree continues splitting until this capacity is reached. There is an inverse relationship between the capacity and the number of leaf cells: a lower capacity results in more cells, while a higher capacity leads to fewer leaf cells. In skewed datasets, the quadtree may become unbalanced, with some branches splitting more frequently. As a result, the final number of partitions is not necessarily a multiple of four. In the figures, we round the number of leaf cells to the nearest thousand."***

**Section 7.5:**

**16. "We can see that the kd-tree takes more time, particularly because of the sorting done at each split, to organize the data and localize the middle point." In the average case, this is indeed a true statement. It would be more interesting to the reader if this could be presented as a percentage of the total runtime.**

In the revised manuscript, we provided additional insights into these percentages. On average, we found that the Quadtree takes 23.13% of the time required to create the Kdtree. Additionally, creating the Kdtree partitions takes, on average, 5.86% of the total DCEL construction time. The following content has been added to Section 7.5:

***"On average, the Quadtree takes 23.13% of the time required to create the Kdtree (21.55% in MainUS and 24.72% in GADM). However, Kdtree creation accounts for only 5.86% of the total DCEL construction time (6.88% in MainUS and 4.87% in GADM)."***

**17. Fig. 23 - Is the x-axis correct? Were the experiments carried out with $21 * 10^6$ cells?**

Thank you for bringing this to our attention. The issue has been corrected in ***Figure 26***.

**Sections 7.6 and 7.7:**

**18. "for datasets with close or similar cardinalities, the area of the dataset has a positive correlation with the build time". The experimental setup needs a better explanation. There are 12 nodes, of which executors are varied from 7 to 84.**

**a. What is the count of executors per node?**

The experiments in Sections 7.6 and 7.7 use a different configuration. In the revised manuscript, we provided more details about the experimental setup and clarified the distinction between physical nodes and executors. To increase parallelism, we divided the 12 worker nodes into 84 worker executors. Each executor is a separate JVM process with its own dedicated resources, such as memory and CPU cores. The distribution of these executors across nodes depends on how the resource negotiator (YARN) allocates resources for Spark jobs, based on available cores and memory. While YARN typically balances resources across the cluster, some variation may occur based on resource availability at runtime. The following content has been added to Section 7.6.

*"We implemented our polygonization framework on Apache Sedona. The experiment is based on a Java 8 implementation and utilizes a Spark 2.3 cluster with two driver nodes and 12 worker nodes. All nodes run Linux CentOS 8.2 (64-bit). Each driver node is equipped with 128GB of RAM, while each worker node has 64GB of RAM. To increase parallelism, we divided the 12 worker nodes into 84 worker executors. Each executor is a separate JVM process with dedicated resources, such as memory and CPU cores. The distribution of these executors across the nodes is managed by the resource negotiator (YARN), which allocates resources for Spark jobs based on the availability of cores and memory. YARN typically balances resources across the cluster, so executors are likely to be evenly distributed, though some variation may occur due to resource availability at runtime. Assuming an even distribution, each worker node would run approximately 7 executors, as calculated by* $\frac{84}{12} = 7$*."*

**b. In the base case, why is there no one-to-one correspondence of nodes to executors (12:7)?**

In this experiment, we utilized virtual executors, with a total of 84 virtual executors on the 12-node cluster. We demonstrated the impact of adding 7 executors, which is roughly equivalent to adding one additional node under an even distribution. The following content has been added to Section 7.7:

*"At each step in the figure, we add 7 more executors, which is approximately equivalent to adding one additional node."*

**c. Possible typo in 10.2M** $Mkm^2$

This typo has been corrected in Section 7.6 of the revised manuscript.

**Section 7.8:**
**19. The section will benefit from stats such as the number of dangles and cut edges generated/resolved.**

We updated **Table 9** to include statistics on the number of polygons in the first layer, the number of dangle and cut edges from the second layer, and the resulting number of polygons after the overlay. The final polygon count provides insight into how many of the input dangles and cut edges intersect with the polygons from the first layer.

**References:**
**21. Please maintain the capitalization style of names and approaches in the paper titles.**

Thank you for your feedback. In the revised manuscript, we corrected the formatting styles in the reference list.