Multi-Step Processing of Spatial Joins

The spatial join combines spatial objects from a Cartesian product that satisfy certain geometric attributes. Unfortunately, to validate the operation the actual objects must be compared against each other. The authors discussed a new process for reducing the number of items that have to be retrieved from disk and compared through filtering and approximation. The three step pipeline started by using the MBRs of objects within an R-Tree to create an approximation of the join set. The second step filtered the candidates with various tricks, reducing the total number that needed to be checked with their actual geometries. The third step performed the actual checking, but was sped up through better representation of data.

The authors assumed that they would only deal with geometries stored in 2-dimensional points and that intersection joins were the only joins, limiting their work to a small degree. Their major contribution was a move away from the naïve nested loop approach used in spatial joins (the approach checked the geometries of every possible object). The new approach accelerated the expensive steps through filtering and reduction of the datasets. As mentioned, a spatial access method was used to access the MBRs of the geometries. Joining these objects created candidate pairs that still needed to be verified. The MBR-join (performed in the R-Tree) was a fast, but inaccurate filter, since it only stored four points for each area. Their measurements showed that 1/3 of the MBR-join candidates were actually false positives.

To reduce the number of false positives, the second step used multiple, alternative approximations of the geometries. A number of convex approximations were explored, but the authors chose the 5-corner minimum bounding polygon as it gave the best performance/memory storage requirement/CPU processing ratio. The 5-C method detected 68 percent of the false hits. The authors used further refinements in the phase to confirm true candidates. The false area/overlap test checked to see if the area of the overlap from the approximation was greater than the sum of the padded area the approximation created. If so, then the item was clearly a good candidate. Progressive approximations were used which defined geometries that maximized a rectangle within the actual object. If those areas overlapped, then once again the candidate was valid. These methods improved the performance slightly.

The final step was to test the actual geometries of the remaining unconfirmed candidates, the most time consuming step. The plane sweeping technique was explored here, but they found that if they decomposed the object into a TR-Tree (an R-Tree that stored trapezoids), then performance was improved for larger datasets (creating the TR-Tree had a high initial overhead cost). Descending the TR-Tree was use to determine whether the items intersected. It’s performance gains came from storing large objects on disk in sequential order much like B and R-Trees. The final algorithm improved performance by a factor of three over naïve solutions.
Partition Based Spatial-Merge Join

Joins are a cornerstone of any relational database and are just as important in spatial data. Combinations of different sets are useful for operations like map overlays or when disparate items with common attributes need to be viewed together. Such operations consist of two steps: filtering and refinement. Filtering uses a coarse representation to remove obvious tuples that are not a part of the join and refinement looks at the actual objects in the remaining set to determine the true answer. Both of these steps are further helped or hindered by the presence of indexes on one, both or neither of the tables.

The authors proposed a new algorithm for joining spatial data that tackled both filtering and refinement. Previous algorithms usually focused only on the filtering step. Some required transforming spatial data into more familiar dimensions, while others only worked well if indexes were present. The partition based spatial merge join algorithm attempted to define an effective method for indexed and non-indexed data, for both filtering and refinement. In PBSM, the filter step was responsible for pairing tuples from the two tables so that their MBRs overlapped. This was done through a well-known plane-sweeping technique that worked if both sets fit into memory. The contribution of the PBSM was if both sets did not fit into memory. In that case, the total space (the universe) was partitioned into different areas and for each object that had an MBR in the partition, it's ID was added to that partition's list. The plane sweeping was then done on each partition. If the distribution was non-uniform though, then equal sized partitions would not filter well and the join would perform poorly. To solve the issue, they introduced tiling, which partitioned the partitions into smaller segments and assigned items to partitions in a round robin fashion. Dense regions were then more uniformly partitioned and subsequent plane sweeping was easier. The refinement step served two purposes: to remove duplicate entries and to determine if the objects actually satisfied the join. The step had the potential for incurring a large number of random seeks, which was addressed by sorting the IDs and then reading the objects into memory sequentially, which significantly reduced the I/O overhead.

To prove its utility, the authors compared the PBSM algorithm to indexed nested loops and R-Tree joins, two other spatial join methods. The nested loops algorithm worked by building an index on the smaller of the two sets (using bulk loading of data) and then sorted based on the MBR. The smaller set was then checked with the values of R that corresponded. The R-Tree algorithm built indexes on both sets, then joined by doing a tree traversal. The comparative study used real data in a variety of setups. The main lesson was that PBSM was much better when memory was at a premium and slightly better when memory was ample. If there were no indexes on the data it also did better because PBSM did not need to build them. If indexes already existed, then the R-tree did better (but only if the smaller of the sets had the index, otherwise PBSM did better). CPU costs were surprisingly more dominant than I/O costs, which the authors thought could be mitigated with parallelism.