Pipelined Spatial Join Processing for Quadtree-based Indexes

Walid G. Aref
Department of Computer Science, Purdue University
aref@cs.purdue.edu

ABSTRACT

Spatial join is an important yet costly operation in spatial databases. In order to speed up the execution of a spatial join, the input tables are often indexed based on their spatial attributes. The quadtree index structure is a well-known index for organizing spatial database objects. It has been implemented in several database management systems, e.g., in Oracle Spatial and in PostgreSQL (via SP-GiST). Queries typically involve multiple pipelined spatial join operators that fit together in a query evaluation plan. In order to extend the applicability of these spatial joins, they are optimized so that upon receiving sorted input, they produce sorted output for the spatial join operators in the upper-levels of the query evaluation pipeline. This paper investigates the use of quadtree-based spatial join algorithms and how they can be adapted to answer queries that involve multiple pipelined spatial joins in a query evaluation plan. The paper investigates several adaptations to pipelined spatial join algorithms and their performance for the cases when both input tables are indexed, when only one of the tables is indexed while the second table is sorted, and when both tables are sorted but are not indexed.

Categories and Subject Descriptors

H.2. [Database Management]: H.2.4. [Systems]: Query Processing

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Spatial join algorithms, spatial databases, query optimization.

1. INTRODUCTION

Many quadtree-based algorithms for performing spatial join exist in the literature, (e.g., see [3] for a survey). Almost all spatial join algorithms are not designed with *pipelined execution* in mind. One exception is the quadtree spatial join algorithms presented in [4], where the authors propose adaptations to existing quadtree-based spatial join algorithms so that the adapted algorithms fit in a pipelined query evaluation plan. One weakness in [4] is that the authors do not provide any experimental study to demonstrate the performance effectiveness of their proposed adaptations. This paper proposes further optimizations for quadtree-based spatial join algorithms that make such algorithms more suitable for answering complex queries via pipelined query evaluation plans. The paper also studies experimentally the performance of the proposed algorithms and compares them to other existing pipelinable algorithms (the ones in [4]).

Refer to Figure 1 for illustration. In a complex query scenario, multiple spatial join operations need to be performed (e.g., Spatial Joins A and B in Figure 1). Typically, multiple spatial join operators will need to fit in a *query evaluation pipeline*, where the output of one spatial join operator (e.g., Spatial Join A in Figure 1) at the lower level of the query evaluation pipeline will feed into the input of another spatial join operator (e.g., Spatial Join B in Figure 1) at the higher level of the query evaluation pipeline. Many of the existing spatial join algorithms are not designed for pipelined execution. Thus, when fit in a query evaluation pipeline, these algorithms cannot take advantage of many of the features that make them work efficiently. For example, the spatial filter (SF) algorithm [5],[6] assumes that both inputs are indexed. Therefore, SF is only applicable at the leaf level of a query evaluation pipeline, where indexes are available. SF is not applicable at higher levels of the query evaluation pipeline (unless a temporary index is constructed on the intermediate results). For example, in Figure 1, since the output of Spatial Join A is not indexed, SF is not applicable as Spatial Join B since there is no index on its input. Similarly, the spatial merge (SM) join algorithm assumes that both inputs are sorted based on a space-filling curve order. Generally, in a query evaluation pipeline, this property is not guaranteed at higher levels of the pipeline, as it depends on how operators at the lower level in the pipeline produce their intermediate output results. In Figure 1, since the output of Spatial Join A is not sorted, SM is not applicable as Spatial Join B since one of B's inputs is not sorted. Unless different measures are taken, SM would be applicable only at the leaf level of a query evaluation pipeline. Generally, constructing a temporary index on the fly (as mandated by SF) or sorting the data based on a space-filling curve order (as mandated by SM) are costly solutions. So, we are left with nest-loops join, which is also an expensive choice. In this paper, several algorithms are proposed that make use of a quadtree-like index and that operate efficiently when embedded in a query evaluation pipeline. A thorough performance study is conducted for various query evaluation pipelines under a variety of conditions including low and high selectivities at the various levels of the query evaluation pipeline. Several findings and rules of thumb are derived from the experimental results that can guide a query optimizer to generate efficient query evaluation pipelines for query processing in spatial databases.

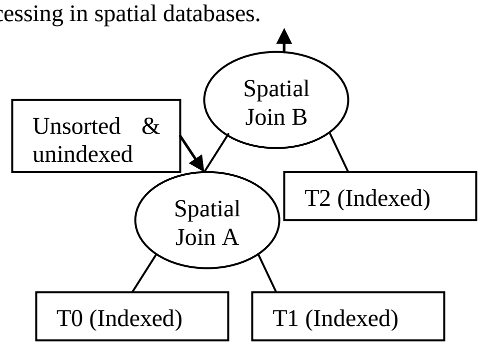


Figure 1 - A sample query evaluation pipeline.

Orenstein and Manola [6],[7] present two algorithms for spatial join (overlap) using the Z-Order as their underlying organization of the spatial database. The first algorithm, termed the *spatial merge* (SM, for short), requires that the two input streams be sorted but does not necessarily require the presence of an index. The second algorithm, termed the *spatial filter* (SF, for short), is an optimization of the spatial merge. The spatial filter yields significant execution time speedups over the spatial merge at the added expense of requiring that both input streams be indexed (e.g., as a result of sorting on the basis of the spatial attribute so that random as well as sequential access to the elements in either stream is possible). The speedups are a direct result of using the index to directly access the input streams.

Aref and Samet [5] present two spatial join algorithms that are optimizations over the spatial merge (SM) and the spatial filter (SF) algorithms. The first spatial join algorithm, termed *linear-scan* (LS), avoids processing every element in the two streams by just scanning the irrelevant intervening elements between corresponding positions in the two streams as it has no index. The second spatial join algorithm, termed *estimate-based* (EB), uses on-line estimates of the input streams to decide whether to use the index for a direct access request or a linear scan.

All the four spatial join algorithms (namely, SM, SF, LS, and EB), require that the input streams be spatially ordered, e.g., according to the Z-order, while the output they produce is not sorted. This is not a problem if the spatial join is only performed once. However, in most applications, it is usually the case that the spatial query requires the execution of a cascade of spatial joins (i.e., the output of one operation serves as input to the next operation). Thus, use of any one of these algorithms means that only the first spatial join in a query evaluation pipeline is efficiently executed. All subsequent spatial joins will require either to spatially sort their input (or build a temporary index) before performing the join, or to use a nested loops join algorithm with a spatial intersection predicate, since none of the above spatial join algorithms works properly for unsorted input streams. In [4], Aref and Samet introduce an algorithmic improvement that is applicable to all the above spatial join algorithms. When this new optimization is augmented to any of the four spatial join algorithms, the augmented algorithm will produce output tuples that are sorted spatially (e.g., using the Z-order sort order). Therefore, any of the modified spatial join algorithms can then be applicable in spatial query pipelines that involve multiple cascaded spatial joins without the need to re-sort the output at each intermediate stage.

In this paper, we extend further in the direction of Aref and Samet [4] to increase the applicability of spatial join filters in query evaluation pipelines. The contributions of the paper are as follows.
1. We propose two new algorithmic adaptations, ½SFu and ½SFs that extend the applicability of the spatial filter algorithms to cases when only one of the two input streams is indexed.
2. We perform an elaborate experimental study to assess the performance gains of the proposed techniques in comparison with the techniques proposed in [4] (There was no performance study in [4] to study the effectiveness of the proposed techniques).

3. The Underlying Spatial Data Organization

In this paper, we focus on quadtree-based spatial databases, where each spatial object is represented by a set of restricted square elements (termed Morton Blocks) that collectively approximate (and cover) the object. These Morton elements [8] result from a recursive decomposition of the space into four square blocks (i.e., a quadtree-like decomposition [1]). For example, Figure 3a and Figure 3b are the Morton elements corresponding to the spatial data sets in Figure 2a and Figure 2b, respectively. The Morton elements are ordered on the basis of their size and the result of mapping the point at the upper-left corner of each block into an integer. This is achieved by interleaving the bits that represent the values of the *x* and *y* coordinates of the two-dimensional point. The result of the interleaving process is termed a *Morton code* [8].
When computing the spatial join of Morton element decompositions of two spatial data sets, we report the pairs of overlapping blocks and then obtain the appropriate object identifiers by use of lookup operations. For example, the spatial join of the Morton elements in Figure 3a and Figure 3b consists of the block pairs (A,I), (F,J), (H,L), and (H,K) corresponding to the object pairs (3,4), (3,5), (3,6), and (3,5), respectively. Notice that because of the nature of the Morton elements and the recursive decomposition process by which they are constructed, any pair of corresponding Morton elements either in the input or output streams are either equal or are contained in one another; however, they cannot overlap without one Morton element being totally contained in another Morton element.

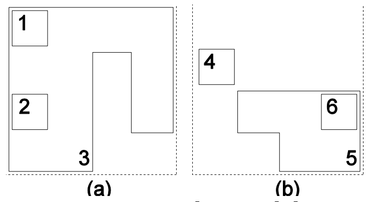


Figure 2 - Two sample spatial data sets.

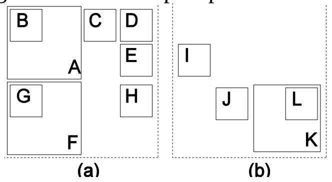


Figure 3 - Result of decomposing the spatial data sets in Figure 2 into their Morton Elements.

2. Spatial Join Algorithms

Nested Loops join (NL) is a traditional join operator that is applicable in the context of spatial databases when neither of the two spatial data sets input to the join are sorted spatially.
When both inputs are sorted but are not indexed, we have several choices, mainly, SMu: the original spatial merge join algorithm with unsorted output [6],[7], SMs: the optimized version of SM that produces sorted output [4], LSu: the original linear scan spatial join algorithm (LS) that produces unsorted output [5], and LSs: the optimized version of LS that produces sorted output [4].

When both inputs are indexed, we can perform an index-only spatial join plan, where the joins are performed by traversing only the indexes. In this category, we have two choices: (1) SFu: the original spatial filter join algorithm (SF) with unsorted output [6],[7], and (2) SFs: the optimized version of SF that produces sorted output [4].

The half spatial filter join operators (½SFu and ½SFs) are the ones we introduce in this paper. For the two tables being joined, if one table is indexed on the spatial attribute involved in the join and the other is not, the spatial filter merge join algorithm cannot be used because of the absence of the index in one of the input tables. In order to utilize the index available on the other input table and still be able to do a spatial filter join, we introduce the half spatial filter merge join algorithm (denoted by ½SFu). In ½SFu, the first table advances in the same way as in SM to get the next block sequentially. The second (indexed) table advances in the same way as in SF. ½SFu uses the current block from the outer table to access the index and decide on the next block it should advance to. ½SFu produces unsorted output. In a straightforward fashion, using the techniques in [4], the half spatial filter join algorithm can be adapted to produce sorted output without additional complexity in its execution (simply, by changing the timing when the output tuples are to be reported). This adaptation results in the half spatial filter join algorithm with sorted output (½SFs). ½SFs is the same as ½SFu except that ½SFs is improved to produce sorted output.

3. Experimental Evaluation

All the experiments are executed on a machine with Intel Pentium IV, CPU 2.4GHZ, and 512MB RAM running Windows XP. In the experiments, we consider query evaluation pipelines that involve three tables, T0, T1, and T2. The tables are generated synthetically by selecting random line segments from the Census Tiger files. For each of the selected lines, a minimum enclosing rectangle is generated. Then, each generated rectangle is decomposed into the Morton Blocks (or region quadtree blocks) that compose the rectangle. For rectangle decomposition, we use any of the window decomposition algorithms, e.g., [9]. We group the experiments into three cases based on the availability of indexes on the tables. In Case 1, all the three tables have indexes on the spatial attribute. In Case 2, T0 and T1 have indexes while T2 does not. In Case 3, none of the three tables has an index on the spatial attribute.

As an example, consider the query evaluation pipeline (QEP) in Figure 4. In this QEP, tables T0 and T1 are both sorted and indexed while T2 is sorted but is not indexed. T0 joins with T1 using a spatial filter merge join. The intermediate resulting table is written to disk and is sorted via an external sort operator. Then, the sorted intermediate table joins with T2 using a spatial merge join, which produces an unsorted final output. Apparently, the behaviors of Cases 1—3 above are related to the sizes of tables T0, T1, and T2 as well as to the sizes of the intermediate join results. The experiments are conducted to address two scenarios: (1) relatively small intermediate output (high join selectivity), e.g., when joining T0 and T1, and (2) relatively larger intermediate output (low join selectivity), e.g., when joining T0 and T1.

3.1 Performance Results for Case 1

In Case 1, all of the three tables have indexes. Figure **5** gives four possible query evaluation plans 1.1—1.4 for spatially joining T0, T1, and T2. In the performance graphs to follow, line# corresponds to the number under each plan in Figure 5 (e.g., line 3 and line 5 correspond to Plans 1.3 and 1.5, respectively).

High Join Selectivity: In this study, T0 has 6201 blocks and T1 contains 6968 blocks. The size of the intermediate result, i.e., T0 join T1, is 343 blocks. The size of T2 varies from around 5,000 to 150,000 blocks. Figure 6a gives the performance results. The x-axis corresponds to the size of T2 while the y-axis corresponds to the running time of the entire query evaluation plan. Lines 1-4 correspond to the performance of Plans 1.1-1.4 in Figure 5.

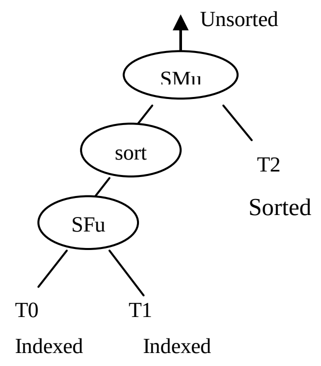


Figure 4. A Sample evaluation plan for spatial join processing.

Plan 1.2 represents the traditional approach to performing multiple spatial joins, i.e., a sophisticated spatial join algorithm (in this case, the spatial filter [6],[7]) being applicable only at the lowest level of the query evaluation pipeline, and then either nest-loops (NL) or nested loops with index (NI) at the higher levels, depending on the presence or lack of an index on the inner table.

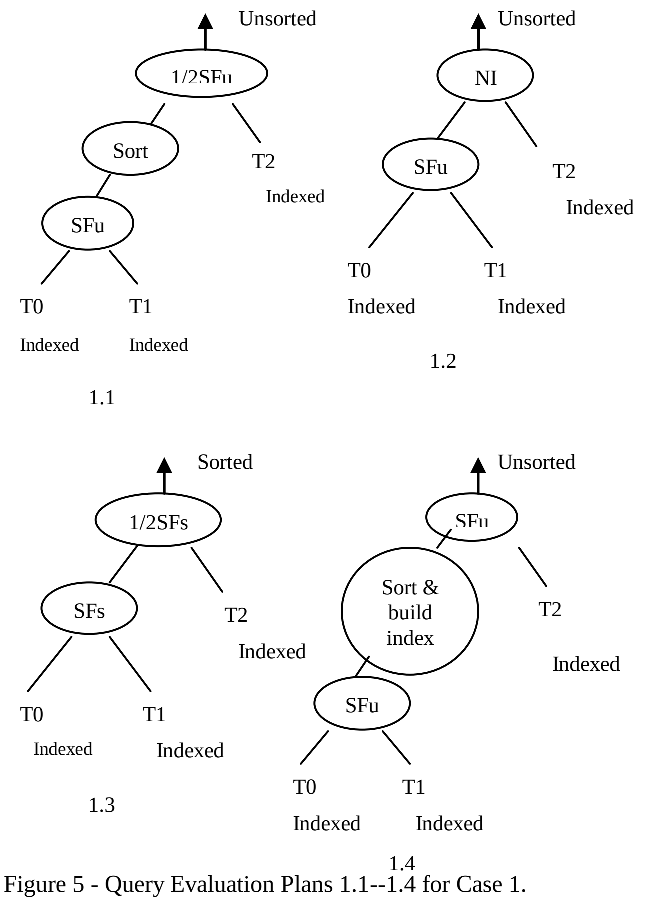
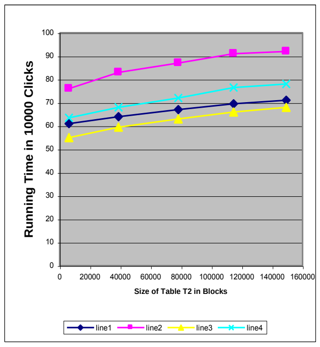
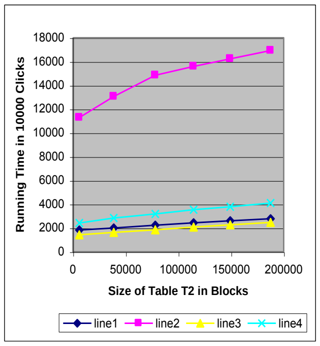


Figure 5 - Query Evaluation Plans 1.1--1.4 for Case 1.

Refer to Figure 6a. Plan 1.2 (line 2 in the figure) performs the worst (takes the longest running time in comparison to all the other plans). Plan 1.4 (line 4 in the figure) is an enhancement over Plan 1.2, where a temporary index is built on the intermediate join result so that the spatial filter join (SFu) can be applicable at the higher level of the query evaluation pipeline. Because of the high selectivity of the lower-level join (the size of the intermediate join result is small), the cost of constructing the temporary index is affordable and pays off. Plan 1.4 performs better than Plan 1.2 but worse than the others. Plan 1.1 (line 1 in the figure) performs better than both Plans 1.2 and 1.4, where Plan 1.1 makes use of the ½SFu join operator, as it only sorts the intermediate join result to prepare it for the next level. Plan 1.3 performs the best among all plans. Plan 1.3 demonstrates the benefits of using all the features, mainly using the SFs spatial filter join operator with sorted output at the lowest level of the pipeline and hence alleviates the need to explicitly sort the intermediate results and ½SFs, in contrast to nested-loops join, at the higher level of the query evaluation pipeline. In summary, in the case of high selectivity, Plan 1.3 is the best plan and can achieve up to 30% enhancement in performance (the enhancement in performance is computed as: 100X(running-time-of-old-plan – run-time-of-new-plan)/ running-time-of-old-plan.





(a) (b)

Figure 6 - Performance Results of Plans 1.1 - 1.4 for Case 1 in the case of (a) high selectivity and (b) low selectivity.

Low Join Selectivity: In this set of experiments, both T0 and T1 have synthetic spatial data sets of size around 100,000 blocks. The size of the intermediate join result (T0 join T1) is around 140,000 blocks. The size of T2 ranges from 6,000 to 200,000 blocks. The results are given in Figure 6b. Similar to the case of high join selectivity in the previous section, Plan 1.3 has the best performance and achieves up to 90% enhancement in performance over Plan 1.2. Comparing Plan 1.3 with Plan 1.1, Plan 1.3 performs 11% to 22% better than Plan 1.1 depending on the size of T2.

Due to space limitations, the experimental results for cases 2 and 3 are omitted from the paper.

4. Concluding Remarks

As evident from the experimental results, the usage of the improved spatial join algorithms is very effective. Since these algorithms produce spatially sorted join output, the upper level join in the query evaluation plan can consume directly the output of the lower level join without the need for writing the intermediate results into disk and without the need for a blocking sort operation. This is true for both cases when the tables to be joined are spatially indexed or not. Another important advantage is that the results of the upper-level spatial join is also sorted without additional cost, and hence can be utilized in further pipeline operations.

The indexes should be exploited as much as possible. ½SFs provides a good method to achieve this. ½SFs is best used as the upper level join. In contrast, in order to be able to use SF at the upper levels, we need to build a temporary index for the output of the lower level joins, which is not often profitable.

In almost all cases, nested loops join results in bad performance. However, for high selectivity of the lower-level join, SFu+NI (the second join is an indexed nested loops join) has an execution cost that is close to that of SFs+½SFs. However, SFs+½SFs has the advantage of producing sorted final result. For low selectivity of the first join, the cost of SFu+NI is high, which makes it not a suitable choice.

In the cases when there is no index available in the lower or upper level joins or in the second join, LS spatial join always has better performance, especially when the first two tables being joined are the ones smaller in size. So, LSs+LSs will have more apparent advantage when the first join has high selectivity than when it has low selectivity.

5. Acknowledgements

This research was supported in part by the National Science Foundation under Grant IIS-0093116. The author acknowledges the help of Ming Lu in helping with the experiments.

6. REFERENCES

[1] H. Samet, *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[2] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

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

[4] Walid G. Aref and Hanan Samet, Cascaded Spatial Joins, The 4th ACM International Workshop on Advances in Geographic Information Systems (ACM-GIS), pp. 17--24, Rockville, Maryland, December 1996.

[5] Walid G. Aref and Hanan Samet, The Spatial Filter Revisited, The 6th International Symposium on Spatial Data Handling, Edinburgh, Scotland, UK, September 1994.

[6] Jack A. Orenstein, Frank Manola, PROBE Spatial Data Modeling and Query Processing in an Image Database Application. IEEE Trans. Software Eng. 14(5): 611-629, 1988.

[7] Jack A. Orenstein, Spatial Query Processing in an Object-Oriented Database System. SIGMOD Conference, pp. 326-336, 1986.

[8] G. M. Morton, A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing, IBM Ltd., Ottawa, Canada, 1966.

[9] Guido Proietti, An Optimal Algorithm for Decomposing a Window into Maximal Quadtree Blocks, Acta Informatica, 36(4): 257-266, 1999.