Integrated Query Processing Strategies for Spatial Path Queries *

Yun-Wu Huang
University of Michigan
ywh@eecs.umich.edu

Ning Jing†
Changsha Institute of Technology
jning@eecs.umich.edu

Elke A. Rundensteiner
University of Michigan
rundenst@eecs.umich.edu

Path search and spatial query processing have been treated thus far as two disjoint areas by the database research community. However, to process a path query such as "*Find the shortest path from my home to the ski resort that only goes through areas with an altitude < 1000 feet*" requires database support in both path computation and spatial query processing. While support for such *spatial path queries* offers valuable services important to applications such as GIS navigation systems, urban planning, and traveler information systems, query processing and optimization of these types of queries is as of now an unaddressed problem. In this paper, we propose a set of strategies for processing this new class of path queries based on varying design decisions along two dimensions. The two dimensions are (1) the spatial overlap operations between areas and links can be preprocessed or intermixed with the path search process and (2) areas satisfying the query constraint can be prefiltered or dynamically selected during path search. We then propose efficient techniques for implementing the resulting four integrated query processing strategies, utilizing technologies such as spatial joins, R-tree for spatial overlap search, and spatial clustering for efficient path search. In this paper, we conduct extensive experiments for our proposed strategies and evaluate their performance based on I/O efficiency. Our experimental studies produce several interesting observations. For example, we find that (1) preprocessing spatial queries is advantageous for smaller buffer sizes, and a low area selectivity factor even when the preprocessing cost is included, and (2) intermixing path search with spatial queries works better with larger buffer sizes, and a high area selectivity factor.

Index Terms — Spatial Query Processing, Complex Query Processing, Path Finding, Performance Evaluation.

Area: Specialized Database Systems

* This work was supported in part by the University of Michigan ITS Research Center of Excellence grant (DTFH61-93-X-00017-Sub) sponsored by the U.S. Dept. of Transportation and by the Michigan Dept. of Transportation. N. Jing was supported in part by the State Education Commission of P.R. China .

† This work was performed while the author was at the University of Michigan.

1 Introduction

Path query processing is required by many applications that include for example Geographic Information Systems (GIS), navigation systems, urban planning, and traveler information systems [6]. Although simple queries such as "*Find the shortest path from A to B*" are essential services offered by such applications, there is a need for more *customized path queries* with embedded constraints that expand services to satisfy more diverse market demands. In this paper, we study one class of such customized path queries characterized by the embedded constraints resolvable by *spatial queries*. Examples of *spatial path queries* are:

Q1: "*Find the shortest path from A to B that only goes through areas with altitude < 1000 feet.*"

Q2: "*Find the shortest path from A to B that only goes through areas with no flooding history.*"

Q3: "*Find the fastest route from A to B that does not go through potential mine fields.*"

The **Q1** path query is an example of the services required by both navigation systems and traveler information systems. **Q2** can serve as a powerful tool for urban road and highway planning, while **Q3** provides important functions crucial to military movements.

The processing of *spatial path queries* requires the integration of techniques for path finding and for spatial query support. "Integration" here means that spatial query processing needs to be part of the path finding process in order to identify road segments[1] that satisfy (or dissatisfy) the spatial constraint. Since a shortest path is an aggregate of road segments, the *spatial path query* processing can satisfy the embedded spatial constraint by deploying a constrained path search that does not include the invalid road segments in the computation of the shortest path. This paper investigates the strategies of integration between spatial queries and path finding that best optimize the *spatial path query* processing under given circumstances.

To process *spatial path queries*, we first identify that spatial overlap operations could be preprocessed before path search to derive a materialized relation between all overlapping road segments and areas. Alternatively, they could be dynamically invoked during path search to determine the validity of each traversed individual road segment. We call this choice "Link Preprocessing Dimension". Independently from the "Link Preprocessing Dimension", areas that satisfy (or dissatisfy) the embedded constraint, i.e., areas with *altitude* $\geq$ *1000 feet*, can be filtered[2] out to create a smaller search space for spatial query processing. Alternatively, the entire areas can be the search targets for such processing. We call this choice "Area Filtering Dimension".

[1] A road segment is the part of a road between two neighboring intersections. Based on graph theory, we model the intersections as vertices (or nodes), and road segments as edges (or links).

[2] Filtering can apply to areas satisfying or dissatisfying the embedded constraint. In our examples, we assume the number of areas dissatisfying the constraint is significantly smaller than that of the satisfying areas. Therefore we will only discuss the dissatisfying case. The satisfying case is just the inverse of the dissatisfying case.

In this paper, we present various optimization strategies for *spatial path query* processing based on the combinations of decisions on the "Preprocessing" and "Filtering" dimensions. We also propose efficient techniques for implementing the resulting integrated query processing strategies, utilizing technologies such as spatial joins [5], R-tree [9] for spatial overlap search, and spatial clustering [11] for efficient path search. We then conduct extensive experiments running *spatial path queries* on real GIS maps, and evaluate the performance of our proposed strategies based on I/O efficiency.

Previous related research has focused on solutions for either path finding [1, 10, 11, 12] or for spatial processing [4, 5, 8, 9, 13, 16, 19], our goal in this paper is to integrate existing technologies in both areas (such as spatial join, R-tree, spatial clustering, etc.) to effectively address the combined problem. To our knowledge, there has not been any prior research on *spatial path queries* by the database community.

The contributions of this paper are: First, we identify a new class of query problems. Second, we develop a set of solutions with approaches characterized by different combinations of design decisions along the two dimensions of "Preprocessing" and "Filtering". Third, we propose query processing strategies for these solutions based on state-of-the-art DB technologies, such as spatial clustering, spatial joins, etc. Fourth, we conduct extensive experiments evaluating the integration strategies by varying criteria such as buffer sizes, percentages of areas dissatisfying the spatial constraint, lengths of paths, etc. While our proposed strategies and the experimental results are directly useful to applications such as traveler information systems and navigation systems, this paper raises an important database research issue, namely complex query optimization. The two decision dimensions ("Preprocessing" and "Filtering") of our complex query processing methodology and their combinations presented in this paper can be viewed as general optimization strategies for complex queries with non-trivial embedded constraints. As stronger analytical power is demanded from databases in order to process more sophisticated queries, we believe the ability to achieve complex query optimization will become an important asset in new-generation databases. Therefore, while this paper presents optimization strategies for one class of complex queries, similar strategies can be established for other types of complex queries.

This paper is organized as follows. In Section 2, we describe the background on spatial queries and path search, while in Section 3, the proposed four strategies are formally presented. We compare the cost of each strategy in Section 4 and describe the experimental results in Section 5. Related research and conclusions are given in Sections 6 and 7, respectively.

2 Background on Spatial Query and Path Search Support

2.1 Representation of the Spatial Map Data

To process a *spatial path query* such as **Q1**, a database must at least store two types of spatial objects: (1) the spatial properties of all road segments and (2) the spatial properties of all areas. We now establish the data structures that model the segments and areas as assumed throughout the remainder of this paper:

***SEGMENT* table.** *SEGMENT* stores all road segments of the underlying map. Each *SEGMENT* tuple models a road segment with attributes such as the speed limit, length, name of the road it is part of, etc. We use $< i, j, ---, \&o_a >$ to represent a *SEGMENT* tuple, where i is the *origin* and j is the *destination* of the segment, $---$ represents miscellaneous attributes associated with each tuple, and $\&o_a$ represents the reference to the spatial property describing the road segment. The latter is not embedded in the *SEGMENT* table since it is potentially very large and also has variable length for different segments.

***AREA* table.** *AREA* stores all areas of interest in the underlying map. Each *AREA* tuple models an area with attributes such as the size, altitude, the city or township it belongs to, etc. We use $< a, altitude, ---, \&o_p >$ to represent a *AREA* tuple, where a is the area identifier, *altitude* is the altitude in feet, $---$ represents miscellaneous attributes associated with each tuple, and $\&o_p$ is the reference to the area a's spatial property.

***ARC* file.** *ARC* stores the *spatial property* of all road segments. To encode the spatial property (shape) of a road segment, we use an integer followed by a vector of spatial points. The integer represents the total number of points whereas each point in the vector is represented by its x- and y- floating point values of the global coordination system.

***POLYGON* file.** *POLYGON* stores the spatial description of all areas. The encoding of the spatial shape of an area is the same as that for a segment, except that, to qualify for a polygon-shaped area, the first spatial point and the last spatial point must be the same in the encoding.

Figure 1 illustrates the two relationships between the *SEGMENT* table and the *ARC* file, and between the *AREA* table and the *POLYGON* file.

2.2 Path Search Support

In this paper, we model the topology of a map using a *link table* which is a collection of links in the map. Each tuple in the *link table* is represented as $< i, j, w >$ where i is the *source* and j is the *destination* nodes of the link, and w is the link weight. The *link table* can be viewed as a materialized view of the *SEGMENT* table that models the connectivity information with minimum

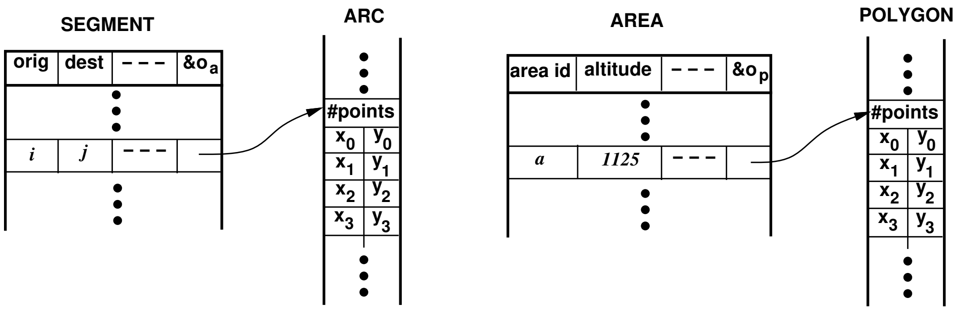

Figure 1: Map Information Configuration.

storage overhead. Path search thus is based on recursive processing of tuples from the more space-efficient link table. We use SP_j^i to represent the shortest path from node i to node j that gives the minimum path weight for all paths from i to j.

Spatial path query processing is based on the shortest path computation for which many well-established single-pair path search algorithms are available in the literature. In this work, we select the *Dijkstra* algorithm because (1) it has an attractive time complexity $O(n \times log(n))$, where n is the number of nodes in graph, for sparse graphs such as GIS road maps, and (2) it exhibits high expansion locality, namely, it searches paths by traversing the links in the graph based on best-found path thus far. Since we assume disk-resident map data, such high expansion locality can be exploited by clustering link tuples based on their spatial proximity.

Because GIS maps typically exhibit high locality[3], clustering link tuples by their spatial proximity guarantees that most nearby links are within the same page. As a result, when the *Dijkstra* algorithm traverses from one link to its connecting links, the number of page misses for retrieving link tuples is reduced. Elsewhere [11], we have shown that such a *spatial clustering technique* results in efficient path search utilizing limited buffer space (in contrast to other alternatives). Thus it lends itself well as a candidate for shared buffer utilization with the spatial processing component of the integrated *spatial path query* processing technique. In this paper, all *link tables* are clustered based on the *spatial clustering technique* proposed in [11].

2.3 Spatial Query Support

We will utilize the following three technologies in spatial query processing:

Spatial Overlap Operator. The spatial overlap operator $\otimes$ that determines if a link overlaps a certain area is implemented based on the *plane-sweep* technique found in [5, 20]. $\otimes$ is a binary operator that takes two spatial property references ($\&o_a$ and $\&o_p$) from a *SEGMENT* tuple and

[3]High locality means the two end nodes of each link are geographically closely located.

an *AREA* tuple respectively as input. It returns *TRUE* if the road segment overlaps the area, and *FALSE* otherwise.

Spatial Search Index. To efficiently retrieve areas that overlap a road segment, we use the *R*-tree index technique with quadratic split [9] because it is the most widely deployed spatial index in existing databases.

Spatial Join Technique. We implement spatial overlap preprocessing using the spatial join techniques proposed in [4] with optimizations achieved in both CPU and I/O [4]. We choose such a technique because it is based on the industry *de facto* *R*-tree structure.

3 *Spatial Path Query* Processing

This section discusses our proposed integration strategies for processing *spatial path queries* using the **Q1** path query (Section 1) as the underlying example.

3.1 The Integration Dimensions Considered in Our Strategy Design

In this paper, we consider two integration dimensions in our strategy design, namely, (1) whether or not to *preprocess spatial overlap* queries to predetermine the overlap relation between links and areas and (2) whether or not to *filter areas* that dissatisfy (satisfy) the spatial constraint.

Decision 1. Preprocess Spatial Overlap: *Spatial Search* (*SS*) vs. *Spatial Join* (*SJ*)?

In processing query **Q1**, the validity of each link that can potentially be included as part of the shortest path must be verified based on the embedded spatial constraint. To verify the validity of a link, one must conduct spatial overlap query processing to select all areas with which it overlaps. A link is valid if all of its overlapping areas satisfy the embedded constraint ("*altitude* < 1000 *feet*"). If such spatial overlap query processing is conducted *during* path search for each link being traversed, we call this approach *Spatial Search*.

Alternatively, if this is conducted *before* path search, we call it *Spatial Join*. In *Spatial Join*, the overlap relation between all road segments and areas is predetermined by preprocessing the spatial overlap queries on the *SEGMENT* and *AREA* tables. The preprocessing step creates a table that materializes the overlap relation. During path search, the validity of the traversed link is determined by simply retrieving its overlapping areas from the structure created by preprocessing without invoking any spatial processing.

Decision 2. Filter Areas Dissatisfying Spatial Constraint: *Filtering* (*F*) vs. *No Filtering* (*NF*)?

So far, we assume spatial overlap query processing required in computing **Q1** searches for the overlapping areas for a given link from the entire set of all areas. We call this approach *No Filtering*. Alternatively, we can filter out all areas that dissatisfy the embedded constraint ("*altitude* $<$ 1000 *feet*") before path search, and store them in a separate structure. This way, spatial overlap query processing is conducted on the specialized (smaller) structure instead of the *AREA* table by retrieving only overlapping areas that dissatisfy the spatial constraint. We call this approach *Filtering*.

There are two potential advantages for *Filtering*. First, the number of areas dissatisfying the spatial constraint may be small, which results in a more efficient spatial search for overlapping areas. Second, the areas stored in the specialized structure are those dissatisfying the spatial constraint. Once one such area is found to overlap a given link, there is no need to retrieve its other attributes, such as *altitude*, to resolve the spatial constraint. Run-time path query performance could improve as a result. However, the specialized structure is dependent on the particular spatial constraint embedded in each *spatial path query*, therefore the cost of creating it must be included as part of the query processing cost. In some cases, the cost of creating the specialized structure can be amortized across several queries that use the same type of spatial constraint.

Combining the alternatives in the above two decisions, we derive four strategies, namely *Spatial Search with No Filtering* (*SSNF*), *Spatial Search with Filtering* (*SSF*), *Spatial Join with No Filtering* (*SJNF*), and *Spatial Join with Filtering* (*SJF*) (see Figure 2).

Preprocessing? / Filtering?	Spatial Search (NO)	Spatial Join (YES)
No Filtering (NO)	SSNF	SJNF
Filtering (YES)	SSF	SJF

Figure 2: The Four Strategies for Processing *Spatial* Path Queries.

3.2 Strategy 1: *Spatial Search with No Filtering* (*SSNF*)

The *SSNF* strategy processes spatial search during *each* link traversal on the entire *AREA* table to determine areas that overlap the traversed link. Then, the altitude information of all overlapping areas is retrieved. The traversed link is valid if all overlapping areas satisfy the embedded constraint i.e., "*altitude* $<$ 1000 feet". Figure 3 is an illustration of our implementation of the *SSNF* strategy.

The *SSNF* strategy establishes the following query processing plan:

1. In general, a link tuple that models the edge relation of a directed labeled graph is of the form

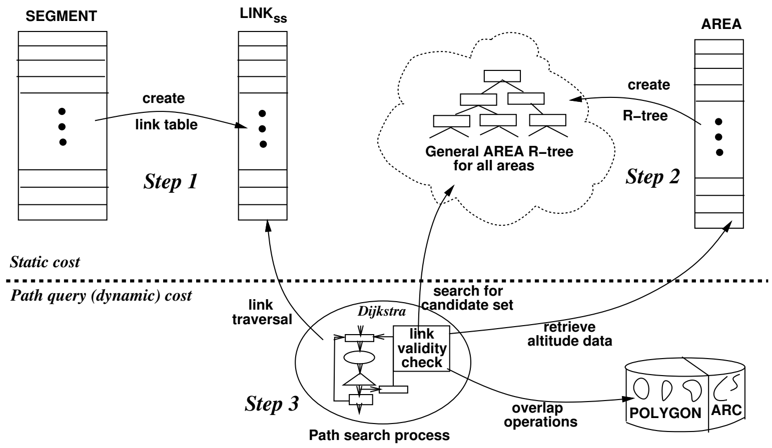

Figure 3: Strategy 1 - *Spatial Search with No Filtering* (*SSNF*).

$< i, j, w >$, where i is the *origin*, j is the *destination*, and w is the link weight. To improve run-time efficiency, the *SSNF* strategy adds two more attributes to its link tuples, namely MBR_{ij} and $\&o_a$. MBR_{ij} is the Minimum Bounding Rectangle (MBR) of the road segment modeled by link $< i, j >$ as required for R-tree operations described in *Step 2* below. $\&o_a$ is the spatial property reference for the respective road segment that is required for spatial overlap operation discussed subsequently. We call the new *link table* $LINK_{ss}$. Adding MBR_{ij} and $\&o_a$ in $LINK_{ss}$ eliminate the need to dynamically compute MBRs and retrieve spatial property refences during *spatial path query* processing, thus improves run-time efficiency. The cost of creating this particular link table $LINK_{ss}$ (*Step 1* in Figure 3) is static which means that it is a one-time[4] cost, therefore is not part of the dynamic query processing cost.

2. An R-tree index is created for the entire *AREA* table (*Step 2* in Figure 3). Such an index improves the spatial search efficiency because searching an R-tree index creates a candidate set of selected areas whose MBRs overlap with the target object's MBR. The computation-intensive spatial overlap operation[5] $\otimes$ is applied on the members from the candidate set instead of from the entire set of areas. Because the spatial overlap operation is computation-intensive in terms of both I/O and CPU time, trading such operations for R-tree index traversal is likely to improve efficiency in both I/O and CPU time. The cost of creating an R-tree for the *AREA* table is considered static because such an R-tree is a general index that can be used by many relevant queries.

3. During the traversal of link $< i, j, w, MBR_{ij}, \&o_a >$ ($< i, j >$ in short) in path search

[4] In real life, the geographic landmarks and street maps may change, but quite infrequently.

[5] They are I/O intensive because the spatial properties of both the segment and the area need to be retrieved from the *ARC* and *POLYGON* files in order to conduct overlap operation. They are CPU intensive because to determine if the two spatial objects actually overlap, the most widely used *plane-sweep technique* [20] requires $O(n \times log(n))$ time complexity, where n is the number of spatial points used to encode the spatial objects.

computation, MBR_{ij} is used to search the $AREA$ R-tree index to generate the candidate set of areas whose $MBRs$ overlap with MBR_{ij}. Next, for each area, say a, the *SSNF* conducts the following checking process. First the altitude data for a is retrieved from the $AREA$ table ($altitude$ in $< a, altitude, ---, \&o_p >$). Next, if $altitude > 1000$ for area a, then the spatial overlap operator $\bigotimes$ is applied between the spatial property reference for the link $< i, j >$ ($\&o_a$ in $< i, j, w, MBR_{ij}, \&o_a >$) and the spatial property reference from a's $AREA$ tuple ($\&o_p$ in $< i, j, altitude, ---, \&o_p >$) to determine if a actually overlaps with $< i, j >$. Lastly, if a does overlap $< i, j >$, $< i, j >$ should not be part of the shortest path, therefore the traversal of $< i, j >$ should be aborted.

Concise Expression:
Find $\{SP_B^A | \forall < i, j > \in SP_B^A, < i, j, w, MBR_{ij}, \&o_a > \in LINK_{ss}, \forall < a, altitude, ---, \&o_p > \in AREA, \&o_a \bigotimes \&o_p = TRUE \longrightarrow altitude \leq 1000\}$.

3.3 Strategy 2 - *Spatial Search with Filtering* (*SSF*)

The *SSF* strategy (see Figure 4) differs from the *SSNF* strategy in that *SSF* first *filters* the $AREA$ table. Namely, it first selects from the $AREA$ table all areas that dissatisfy the query constraint, i.e., *altitude > 1000 feet*, to create a specialized table of areas, denoted by $AREA_{sp}$. The traversed link is valid if it does not overlap with any area in the specialized table. The advantage of such a strategy is that when the number of areas with elevation over 1000 feet is small, the spatial search process becomes more efficient because the size of $AREA_{sp}$ is smaller than that of $AREA$.

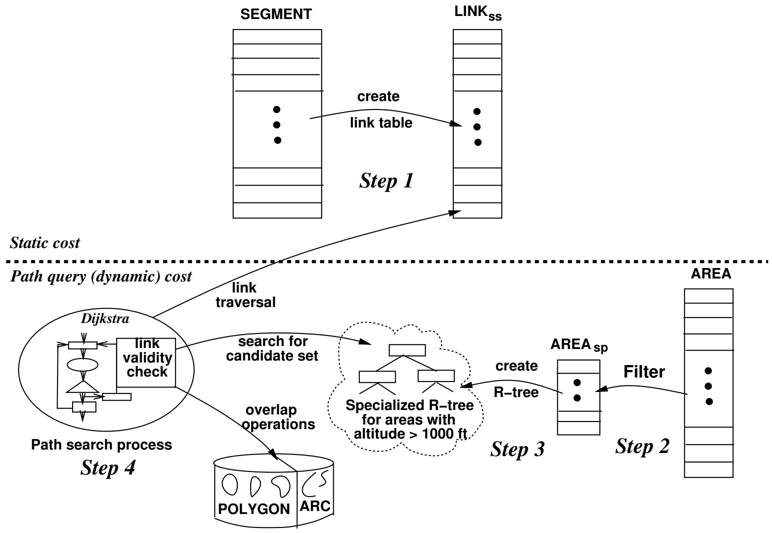

Figure 4: Strategy 2 - *Spatial Search with Filtering* (*SSF*).

Our query process plan for the *SSF* strategy is:

1. The link table $LINK_{ss}$ (same as *Step 1* of the *SSNF* strategy) for the *SSF* strategy must exist, or it is created (*Step 1* in Figure 4). The cost of creating $LINK_{ss}$ is static.

2. Select from the *AREA* table all tuples $< a, altitude, ---, \&o_p >$ with *altitude* > 1000 feet, and store them as a specialized table $AREA_{sp}$ (*Step 2* in Figure 4). The cost of such filtering is considered to be dynamic, meaning that it must be counted as part of the path query processing cost because $AREA_{sp}$ may not be a permanent table in the DBMS.

3. A specialized *R*-tree index is created for the $AREA_{sp}$ table to allow efficient spatial overlap search (*Step 3* in Figure 4). The cost of creating such a specialized *R*-tree is dynamic.

4. During the traversal of link $< i, j, w, MBR_{ij}, \&o_a >$ of the path search process, MBR_{ij} is used to search the $AREA_{sp}$ *R*-tree index to create a candidate set of areas, e.g., $< a, altitude, ---, \&o_p >$, whose *MBRs* overlap MBR_{ij}. Next, the spatial overlap operator $\bigotimes$ is applied between $\&o_a$ and $\&o_p$ of each candidate area to determine if it actually overlaps link $< i, j >$. Link $< i, j >$ is valid only if it does not overlap any area in the $AREA_{sp}$ table, otherwise the traversal of $< i, j >$ is aborted. See *Step 4* in Figure 4 for illustration.

Concise Expression:
Find $\{SP_B^A | \forall < i, j > \in SP_B^A, < i, j, w, MBR_{ij}, \&o_a > \in LINK_{ss}, \forall < a, altitude > 1000, ---, \&o_p > \in AREA_{sp} \longrightarrow \&o_a \bigotimes \&o_p = FALSE\}$.

3.4 Strategy 3: *Spatial Join with No Filtering* (*SJNF*)

The *SJNF* (see Figure 5) strategy first conducts a spatial overlap join between the *SEGMENT* and the *AREA* tables to create a join index. Such a join index stores tuples of the form $< i, j, a >$ where the link identified by $< i, j >$ overlaps the area a. The advantage of doing a spatial join is that the expensive spatial overlap operations during path search are replaced with straightforward search on the join index. The resulting general-purpose spatial join index can be utilized by all queries that require the knowledge of overlap status between road segments and areas. The cost of the spatial join processing thus can be considered static.

The *SJNF* strategy establishes the following query processing plan:

1. The link table $LINK_{sj}$ with link tuples of the form $< i, j, w >$ (see Section 2.2) must exist or be created for path search (*Step 1* in Figure 5). Because spatial queries are preprocessed during spatial join, no additional information is needed in the link tuples. The cost of creating such link table is static.

2. An *R*-tree index must be created for both the *SEGMENT* and the *AREA* table in order to conduct *R*-tree based spatial join using techniques proposed in [4] (*Step 2* in Figure 5). The cost of this step is static because the resulting spatial join index is for general purpose.

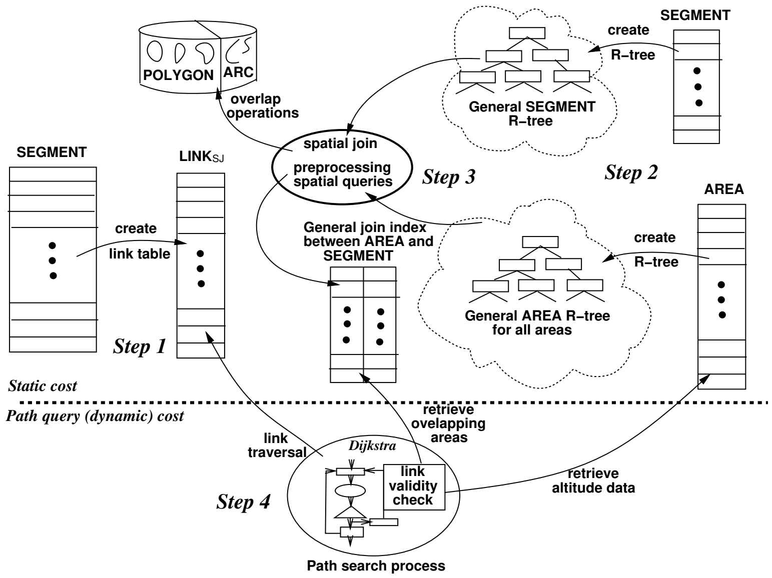

Figure 5: Strategy 3 - *Spatial Join with No Filtering* (*SJNF*).

3. The spatial overlap join is conducted to generate a join index table, called *JOIN_INDEX*, that stores tuples of the form $< i, j, a >$ where the link $< i, j >$ overlaps the area a (*Step 3* in Figure 5). For efficient tuple retrieval, a B-tree is created for the *JOIN_INDEX* (not depicted in Figure 5 for simplicity). The cost of this step is static because the *JOIN_INDEX* is for general purpose.

4. During the traversal of link $< i, j, w >$ of the path search process, $< i, j >$ is used to search *JOIN_INDEX*'s B-tree index. If a join index tuple, e.g., $< i, j, a >$, exists in the *JOIN_INDEX*, the *altitude* data for area a, i.e., *altitude* in $< a, altitude, ---, \&o_p >$, is retrieved from the *AREA* table. Note that we do not store *altitude* information in *JOIN_INDEX* because it is for general purpose, otherwise, the cost of creating such a join index could not be considered to be static. The link $< i, j >$ is valid if no area a in $< i, j, a >$ in the *JOIN_INDEX* has an *altitude* > 1000 feet. See *Step 4* in Figure 5 for illustration of this step.

The Concise Expression:

$JOIN\_INDEX = \{< i, j, a > \mid < i, j, ---, \&o_a > \in SEGMENT, < a, f, ---, \&o_p > \in AREA, \&o_a \bigotimes \&o_p = TRUE\}$.

Find $\{SP_B^A | \forall < i, j > \in SP_B^A, \forall < a, altitude, ---, \&o_p > \in AREA, < i, j, a > \in JOIN\_INDEX \longrightarrow altitude \leq 1000\}$.

3.5 Strategy 4: Spatial Join with Filtering (*SJF*)

The *SJF* strategy (see Figure 6) combines the *Filtering* approach with the *Spatial Join* preprocessing approach. It differs from the *SJNF* in that, instead of creating a join index between all road segments and all areas, it creates a specialized join index $JOIN\_INDEX_{sp}$ between all road segments from the *SEGMENT* table and all areas with *altitude* $>$ 1000 feet from the *AREA* table. Because $JOIN\_INDEX_{sp}$ is dependent on the particular path query constraint, it may need to be recreated for each such path query at the worst case. Therefore, the cost of creating $JOIN\_INDEX_{sp}$ is dynamic.

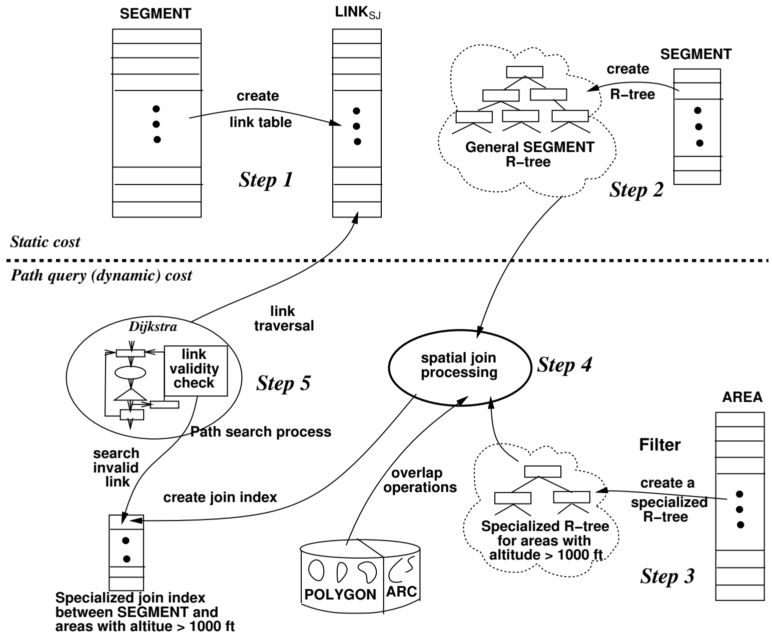

Figure 6: Strategy 4 - *Spatial Join with Filtering* (*SJF*).

The query processing plan for the *SJF* strategy is:

1. Same as in *Step 1* of *SJNF*, the link table $LINK_{sj}$ with tuples of the form $< i, j, w >$ must exist or be created for path search (*Step 1* in Figure 6) with static cost.

2. An *R*-tree index is created for the *SEGMENT* table (*Step 2* in Figure 6) as needed to perform efficient spatial join. The cost of this step is static.

3. Next, we apply *filtering* by selecting from the *AREA* table all tuples, e.g., $< a, altitude, --$ $-, \&o_p >$, with *altitude* $>$ 1000 feet, and create a specialized *R*-tree index for such areas. The cost of creating such a specialized index is dynamic.

4. An *R*-tree based spatial join technique [4] is conducted to create a join index $JOIN\_INDEX_{sp}$ between the *SEGMENT* *R*-tree and the specialized *R*-tree for areas with *altitude* $>$ 1000 feet.

Each tuple in the resulting *JOIN_INDEX*$_{sp}$ is of the form $< i, j >$, indicating that link $< i, j >$ overlaps at least one area with *altitude* over 1000 feet (*Step 4* in Figure 6). In other words, for a particular *spatial* path query, the *SJF* strategy precomputes a complete list of links that are invalid for the path search process. A *B*-tree index is created for *JOIN_INDEX*$_{sp}$ (not depicted in Figure 6 for simplicity) to allow efficient search over *JOIN_INDEX*$_{sp}$. The cost of these operations is considered dynamic.

5. During the traversal of the link $< i, j, w >$ of the path search process, $< i, j >$ is used to search the *JOIN_INDEX*$_{sp}$'s *B*-tree index. The link $< i, j, w >$ is valid if $< i, j >$ does not exist in the *JOIN_INDEX*$_{sp}$. See *Step 5* in Figure 6 for illustration.

The Concise Expression:
$JOIN\_INDEX_{sp} = \{< i, j > \mid < i, j, - - -, \&o_a > \in SEGMENT, \exists < a, altitude > 1000, - - -, \&o_p > \in AREA_{sp}, \&o_a \bigotimes \&o_p = TRUE\}$.
Find $\{SP_B^A | \forall < i, j > \in SP_B^A, < i, j > \notin JOIN\_INDEX_{sp}\}$.

4 Cost Components in *Spatial Path Query* Processing Strategies

$CR_{LINK_{ss}}$	create link table $LINK_{ss}$ for *SSNF* and *SSF* (see Section 3.2)
$CR_{LINK_{sj}}$	create link table $LINK_{sj}$ for *SJNF* and *SJF* (see Section 3.4)
CR_{AREA}^{R-tree}	create *R*-tree from $AREA$ table
$CR_{SEGMENT}^{R-tree}$	create *R*-tree from *SEGMENT* table
$CR_{AREA_{sp}}^{R-tree}$	create *R*-tree from $AREA_{sp}$ table
$SJ_{AREA}^{SEGMENT}$	spatial join between *R*-trees of $AREA$ and *SEGMENT* tables
$SJ_{AREA_{sp}}^{SEGMENT}$	spatial join between *R*-trees of $AREA_{sp}$ and *SEGMENT* tables
$JS_{AREA}^{SEGMENT}$	search join index created by $SJ_{AREA}^{SEGMENT}$
$JS_{AREA_{sp}}^{SEGMENT}$	search join index created by $SJ_{AREA_{sp}}^{SEGMENT}$
RD_{AREA}	retrieve tuples from $AREA$ table
SS_{AREA}^{R-tree}	spatial index search on *R*-tree of $AREA$ table
$SS_{AREA_{sp}}^{R-tree}$	spatial index search on *R*-tree of $AREA_{sp}$ table
$TC_{LINK_{ss}}$	recursively retrieve $LINK_{ss}$ tuples for transitive closure path search
$TC_{LINK_{sj}}$	recursively retrieve $LINK_{sj}$ tuples for transitive closure path search
$WR_{AREA_{sp}}$	create the $AREA_{sp}$ table

Figure 7: Notations of Cost Components.

Each of our proposed strategies for *spatial path query* processing consists of many component operations that incur I/O cost. These components include path search computation, spatial join, spatial index search, spatial index creation, etc. Each operation may access various indexes and table files. This section summarizes and compares the cost components between the proposed strategies. Figure 7 lists the notations of all cost components, both static and dynamic.

4.1 Static Cost

Strategy	$CR_{LINK_{ss}}$	$CR_{LINK_{sj}}$	CR^{R-tree}_{AREA}	$CR^{R-tree}_{SEGMENT}$	$SJ^{SEGMENT}_{AREA}$
$SSNF$	X		X		
SSF	X				
$SJNF$		X	X	X	X
SJF		X		X	

Figure 8: Static Cost Components.

Because static cost is a one-time cost, we may assume that a supportive DBMS may have created the proper tables or access data structures (i.e., the link tables, general R-tree indices, and general spatial join indices) before the actual *spatial path query* processing takes place. Thus, we don't count the static cost as part of the query processing (i.e., dynamic) cost. Figure 8 shows the static cost components for each strategy. Note that some of the static cost components such as $SJ^{SEGMENT}_{AREA}$ may be very expensive and may need extra storage. If precomputation of these structures is not performed by the DBMS, the dynamic query cost for the strategies that rely on these structures may go up dramatically. The query manager may need to discard the strategy that relies on these structures if it detects their absence. For simplicity, we assume that all static cost components are precomputed.

4.2 Dynamic Cost

The dynamic query processing cost can be categorized into three stages: "Filtering", "Preprocessing", and "Path Finding".

Filtering Stage: Both *SSF* and *SJF* strategies filter the $AREA$ table and create a subset table $AREA_{sp}$ which contains only the AREA tuples that satisfy the constraint (i.e., *altitude* < *1000 feet*). Let FI_x be the filtering cost by strategy x, then the cost for filtering in *SSF* and *SJF* is:

$$FI_{SSF} = FI_{SJF} = RD_{AREA} + WR_{AREA_{sp}}$$

The selection cost, RD_{AREA}, can be optimized if an index such as a B-tree exists for the constrained attributes. In our experiments in Section 5 we assume that such an index is not available. The filtering can be done by selecting the tuples that either satisfy or dissatisfy the constraint. If the query manager has the information about the area selectivity factors, proper selection can be made to create a smaller special-purpose filtered table, namely $AREA_{sp}$ in our case.

Preprocessing Stage: Both *SJNF* and *SJF* strategies process the spatial overlap operations in this stage before actual path finding is executed. For *SJNF*, the preprocessing corresponds to conducting spatial join between general R-trees from *AREA* and *SEGMENT* tables. The cost is hence considered static. Therefore, only *SJF* incurs dynamic preprocessing cost. Let PR_x be the preprocessing cost, then the cost for preprocessing in *SJF* is:

$$PR_{SJF} = CR^{R-tree}_{AREA_{sp}} + SJ^{SEGMENT}_{AREA_{sp}}$$

The major cost in this stage is the spatial join, namely $SJ^{SEGMENT}_{AREA_{sp}}$, which includes the cost of joining the two R-trees in creating a candidate set, conducting spatial overlap operations ($\otimes$) to create the answer set, and creating a B-tree index for the answer set. Spatial join performance measurement however remains an open issue to this date. The query manager may need to rely on empirical information to approximate the spatial join cost. If it decides that the preprocessing cost is too high, *SJF* should not be considered.

Path Finding Stage: All four strategies conduct path finding by recursively retrieving the link tuples from link tables for evaluation. While a link is expanded during path search computation, it is checked to see if it satisfies the embedded query constraint. Only the valid links are expanded. The validation process however differs in each strategy. Let PF_x be the path finding cost for strategy x, then we have:

$$PF_{SSNF} = TC_{LINK_{ss}} + SS^{R-tree}_{AREA} + RD_{AREA}$$

$$PF_{SSF} = TC_{LINK_{ss}} + SS^{R-tree}_{AREA_{sp}}$$

$$PF_{SJNF} = TC_{LINK_{sj}} + JS^{SEGMENT}_{AREA} + RD_{AREA}$$

$$PF_{SJF} = TC_{LINK_{sj}} + JS^{SEGMENT}_{AREA_{sp}}$$

Note that *SSF* and *SJF* do not need to retrieve the altitude information by reading the *AREA* table (RD_{AREA}) because all tuples in $AREA_{sp}$ created by the filtering stage satisfy (or dissatisfy) the constraint. They however need to conduct spatial queries (SS^{R-tree}_{AREA} for *SSNF* and $SS^{R-tree}_{AREA_{sp}}$ for *SSF*) that include the cost of searching the R-tree in creating a candidate set, and conducting spatial overlap operations $\otimes$ to create the answer set. *SJNF* and *SJF* do not need to process such spatial queries because the spatial relation between *SEGMENT* and *AREA* has been predetermined and materialized in spatial join indexes. Instead, they incur the cost of searching the B-tree join index ($JS^{SEGMENT}_{AREA}$ for *SJNF* and $JS^{SEGMENT}_{AREA_{sp}}$ for *SJF*).

Total Dynamic Query Processing Cost: The total cost is the sum of the path finding cost plus the filtering and preprocessing costs, if the filtering or preprocessing is conducted. Let DC_x be the total dynamic query processing cost for strategy x, we get:

$$DC_{SSNF} = PF_{SSNF}$$

$$DC_{SSF} = PF_{SSF} + FI_{SSF}$$

$$DC_{SJNF} = PF_{SJNF}$$

$$DC_{SJF} = PF_{SJF} + FI_{SJF} + PR_{SJF}$$

4.3 Empirical Evaluation as A Query Processing Guideline

To establish a cost formula for *spatial path query* processing, cost models for each component operation must be available. Cost modeling for B-tree retrieval is a well-researched area. Recently, cost modeling for R-trees has been proposed [7, 17]. However, to the authors' knowledge, analytical cost modeling for both database transitive closure path search and spatial join is very difficult and remains an open issue to date. To develop a cost model for either operation hence is a subject on its own, and therefore is clearly beyond the scope of this paper. Instead, for performance measurement in *spatial path query* processing, we recommend using empirical evaluation to construct a guideline in selecting the appropriate strategy under certain circumstances.

However, empirical performance measurement for complex queries such as *spatial path queries* is also a non-trivial issue because (1) some cost components share the same buffer resource and (2) the cost components are not mutually independent. We believe deploying combinations of techniques such as heuristic estimation, sampling, and historical data analysis may best approximate the query performance for such complex queries. Although such a difficult task raises many challenging new issues that are not the focus of this paper, it remains an important research question which will be the focus of our future work. In Section 5, we present experimental results that can serve as the foundation for empirical performance measurement for *spatial path query* processing.

5 Experiments

5.1 Experimental Testbed Setup

We have implemented a *spatial path query* testbed using the implementation techniques described in Section 3 on a dedicated SUN SPARC station 20 with 32MB main memory using C/C++. Every set of experiments is repeated ten times with randomized input, and the represented results correspond to an average over these ten runs, respectively.

Due to the overall complexity of spatial query processing, only few analytical results are known for spatial access methods based on the R-tree, and most of these analytical results impose restrictions rarely occurred in real applications [4]. As a result, we investigate the performance of *spatial path queries* based on empirical results with experiments conducted on data sets from real GIS applications. In this paper we focus on I/O performance trade-offs while the CPU time efficiency of these strategies will be presented in an on-going technical report for space limitation reason.

5.2 Experimental Data Set and Parameters

We base our experiments on a very fine-granularity street map of the Ann Arbor City road network which contains 5,596 nodes and 14,033 links. Because we have no area data available, we randomly create spatial areas covering the Ann Arbor City. Given the total number of desired areas, the area map is constructed by randomly generating approximately equal sized areas that are evenly distributed over the Ann Arbor City region.

We assume that all experimental data sets (except a small number of temporary variables) are stored on secondary storage and a limited number of main memory buffer pages of 2KB size each are available (typically ranging from 100 to 250 pages). Our experimental map has less than 15,000 links. For large city maps, the number of links can be many times more than our experimental map[6]. This means that the buffer size should also increase proportionally for large maps in order to effectively process *spatial path queries*. We measure the I/O cost of the *spatial path queries* by varying the following parameters:

- *Buffer Size* is the total number of buffer pages assigned to the testbed.
- *Path Length* classifies any O-D pair into three categories based on direct O-D distance d:
  (1) long paths: $d > 2/3 \times d_{max}$, where d_{max} is the maximum distance between any O-D pair.
  (2) medium paths: $1/3 \times d_{max} < d \leq 2/3 \times d_{max}$.
  (3) short paths: $d \leq 1/3 \times d_{max}$.
- *Area Size* is the approximate size of the areas created by our random data generator, though each area has its own unique variation in shape.
- *Total Number of Areas* is the number of areas generated to cover the Ann Arbor City map.
- *Selected Areas* is the percentage of areas from the set of all areas that dissatisfy the query constraint (See footnote 2).

5.3 Experiment of I/O Cost of *SSNF*

The first experiment evaluates the I/O cost of *SSNF* in relationship to both the *buffer size* as well as the percentage of *selected areas*. In Figure 9, we randomly pick ten O-D pairs of long paths and measure the average I/O cost of *spatial path query* processing for each combination of different *buffer*

[6] For example, the Detroit map we used in a separate research effort has more than 50,000 links.

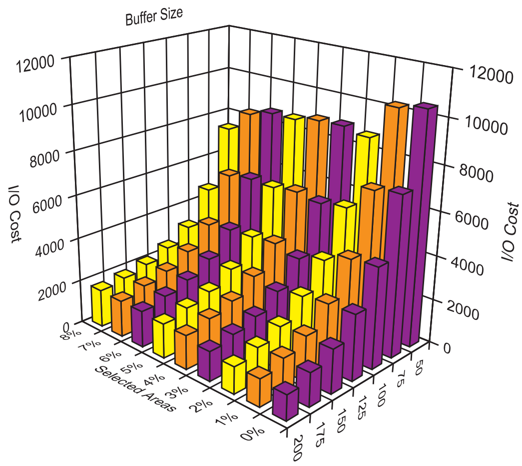

Figure 9: I/O Cost of *SSNF*.

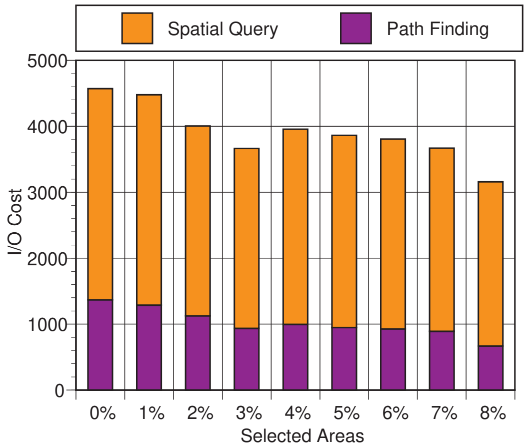

Figure 10: I/O Cost of *SSNF* vs. Selected Area with 100 Buffer Pages.

sizes and *selected areas*. Unsurprisingly, the I/O cost goes down with the increase of *buffer size*. The increase of *selected areas* however does not cause the I/O cost to linearly decrease. The I/O exhibits a wave-shape that goes down, then up, and down again with the increase of *selected areas*. To explain this, we note the *SSNF* intermixes path finding (which incurs $TC_{LINK_{ss}}$) with spatial query processing (which incurs $SS^{R-tree}_{AREA}+RD_{AREA}$). $TC_{LINK_{ss}}$ depends on the number of links expanded during path finding, whereas $SS^{R-tree}_{AREA}+RD_{AREA}$ may increase as *selected areas* increase. However, the higher the *selected areas*, the fewer the valid links that satisfy the query constraint meaning there are fewer links available for path search expansion. Consequently, $TC_{LINK_{ss}}$ may go down when *selected areas* increases. While $SS^{R-tree}_{AREA}+RD_{AREA}$ may go up because there are more areas satisfying the spatial queries, it also may go down because fewer link expansions may cause fewer such queries that need to be processed. Such a conflicting effect contributes to the wave-shape in Figure 9.

To verify our analysis, we designed another experiment which separates the I/O cost into two parts, namely the *path finding* cost and *spatial query* cost. The experimental results (Figure 10) confirm our analysis.

5.4 Experiment of I/O Cost of *SSF*

In this experiment, we measure the total I/O cost of *spatial path query* processing for the *SSF* approach by varying the *buffer size* and the percentage of *selected areas* (Figure 11). As expected, the I/O cost decreases again with the increase of the *buffer size* since the larger the *buffer size*, the less page swapping occurs. The I/O cost goes up with the increase of *selected areas*, which is different from the situation in *SSNF* (Section 5.3). The total I/O cost of *SSF* consists of three parts: *filtering* cost ($RD_{AREA} + WR_{AREA_{sp}}$), *path finding* cost ($TC_{LINK_{ss}}$), and *spatial query* cost

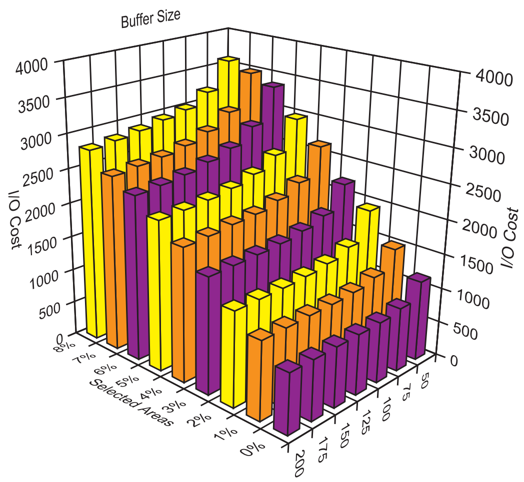

Figure 11: I/O Cost of *SSF*.

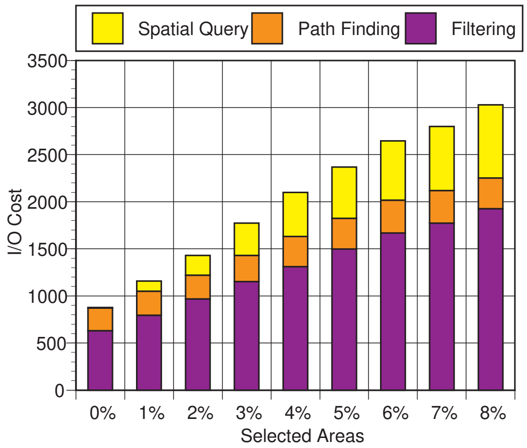

Figure 12: I/O Cost of *SSF* vs. Selected Area with 100 Buffer Pages.

($SS^{R-tree}_{AREA_{ss}}$). In Figure 12, we again separate these costs by fixing the *buffer size* to 100 and varying the *selected areas*. Among these three kinds of costs, the filtering cost is the most expensive one and the path finding the least expensive. Because the specialized R-tree grows with the increase of the percentage of *selected areas*, the costs of both spatial query and filtering increase accordingly.

5.5 Experiment of I/O Cost of *SJNF*

Figure 13 shows the I/O costs of *SJNF*. The cost decreases with the increase of the *buffer size*. Like *SSNF*, *SJNF* also deploys an intermixed strategy that combines path finding with spatial queries that search the general join index. A conflicting effect similar to that of *SSNF* therefore may exist that causes a wave-shape result. To confirm this, we separate the total cost into the *spatial query* cost ($JS^{SEGMENT}_{AREA} + RD_{AREA}$) and the *path finding* cost ($TC_{LINK_{sj}}$). Figure 14 depicts the separated costs by varying the percentage of *selected areas*. The pattern of change confirms our prediction.

5.6 Experiment of I/O Cost of *SJF*

Figure 15 shows the relationship of the I/O cost vs. the *buffer size* and *selected areas* for the *SJF* strategy. The I/O cost decreases when the *buffer size* decreases. Similar to the *SSF* approach, the total I/O cost consists of three parts: *overhead* cost ($RD_{AREA}+WR_{AREA_{sp}}+CR^{R-tree}_{AREA_{sp}}+SJ^{SEGMENT}_{AREA_{sp}}$)), *path finding* cost ($TC_{LINK_{sj}}$), and *spatial query* cost ($JS^{SEGMENT}_{AREA_{sp}}$). The *overhead* cost here is the total of the "Filter" cost and "Preprocess" cost described in Sections 3 and 4. Figure 16 depicts the separated costs by fixing the *buffer size* to 100. Note that the specialized join index created by the preprocessing step in *SJF* grows dramatically in size with the increase of the percentage of *selected areas*. This is because of the spatial join "product" effect caused the increased number of

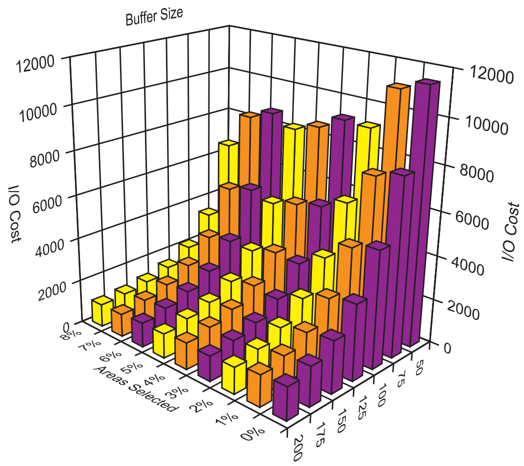

Figure 13: I/O Cost of *SJNF*.

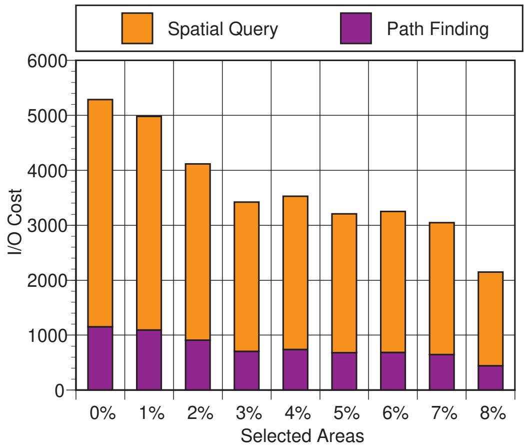

Figure 14: I/O Cost of *SJNF* vs. Selected Area with 100 Buffer Pages.

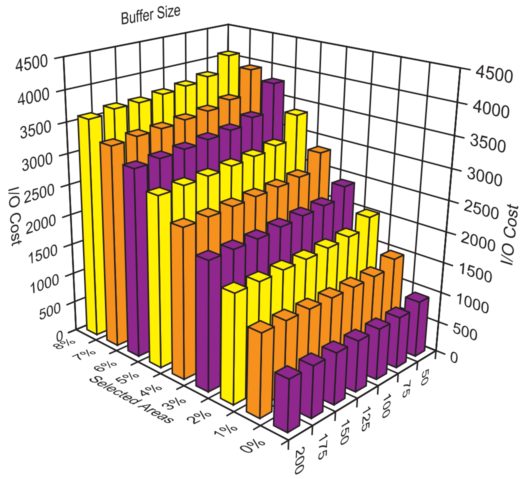

Figure 15: I/O Cost of *SJF*.

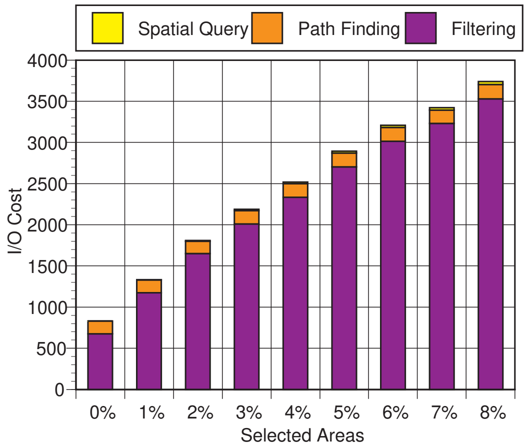

Figure 16: I/O Cost of *SJF* vs. Selected Area with 100 Buffer Pages.

the *selected areas*. As a consequence, the *overhead* cost increases dramatically with the increase of *selected areas*. The *spatial query* cost is very small compared to the cost of filtering since the most expensive spatial computation has been done in the filtering process. In fact, the *spatial query* cost in this case is the lookup of the small specialized join index which contains only information of those links that overlap the *selected areas*. The *path finding* cost is also small compared to the *filtering* cost because it only needs to retrieve the relatively small connectivity information from the properly clustered link table.

5.7 Experiment of I/O Cost vs. the Buffer Size

The purpose of this experiment is to compare the relative I/O efficiency of the four *spatial path query* strategies with respect to the changes of the *buffer size*. In this experiment, we create 10,000 areas, fix the *selected areas* to be 5% of the total areas, and measure the I/O costs of the *spatial*

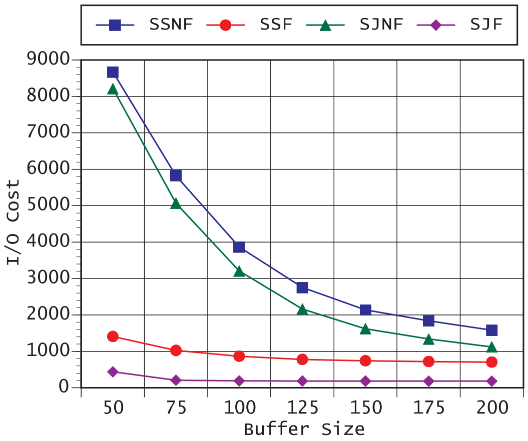

Figure 17: I/O Cost vs. Buffer Size (Without Filtering Cost).

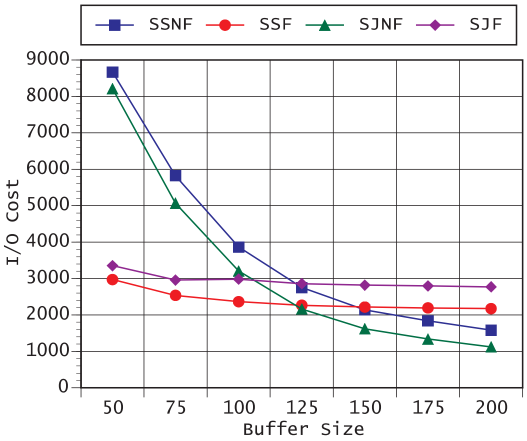

Figure 18: I/O Cost vs. Buffer Size (With Filtering Cost).

path query for long paths. Figure 17 does not count the dynamic filtering costs of the two filtering strategies, namely *SSF* and *SJF*, while Figure 18 includes the filtering costs. In Figure 17, the I/O costs of all four strategies decrease with the increase of the *buffer size*. Compared with *SSF* and *SJF*, the I/O costs of *SSNF* and *SJNF* go down sharply with the increase of the *buffer size*. This is because the general *R*-tree of *SSNF* is much bigger than the specialized *R*-tree in *SSF*, thus more page swapping is likely to occur for *SSNF* (but not for *SSF*) in the case of smaller *buffer size*. Therefore, with the increase of the *buffer size*, the I/O cost of the *SSNF* reduces more than that of the *SSF*. A similar reason explains the improvement of the I/O cost of *SJNF* over *SJF* when increasing the *buffer size*. Among these four strategies, the *SJF* has the least I/O cost when discounting the filtering cost because the *SJF* strategy does all the expensive spatial query processing in the filtering process. Counting the filtering costs, Figure 18 shows that the *SJF* now becomes consistently more costly than the *SSF*. This indicates that the filtering cost of *SJF* is higher than that of *SSF*. This is because the filtering process of the *SSF* involves only generating the specialized *R*-tree, while the filtering process of the *SJF* consists of two steps: first, it creates the specialized *R*-tree; then it uses this specialized *R*-tree in the spatial join operation to generate the specialized join index. Figure 18 also indicates that for smaller *buffer sizes*, the *SSF* is preferred, while for bigger *buffer sizes*, the *SJNF* gives better performance.

5.8 Experiment of I/O Cost vs. the Selected Areas

In this experiment, we compare the I/O efficiency of the four strategies by varying the percentage of *selected areas*. Fixing the *buffer size* to 100, we measure the I/O cost of each *spatial path query* strategy for long paths by varying the percentage of *selected areas* out of 10,000 areas. Figure 19 does not include the filtering cost for *SSF* and *SJF* strategies, while Figure 20 does. With the increase of *selected areas* in Figure 19, the costs of *SSNF* and *SJNF* decrease, while the costs of

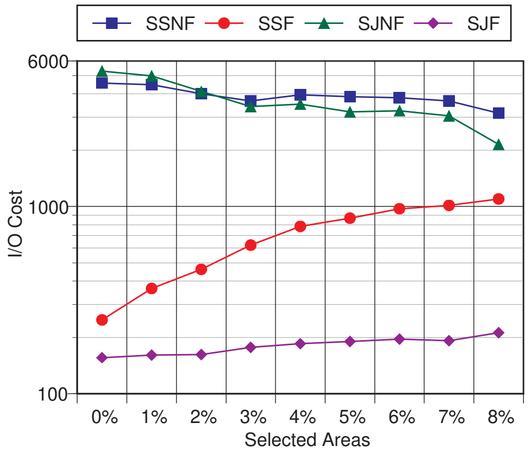

Figure 19: I/O Cost vs. Selected Areas (Without Filtering Cost).

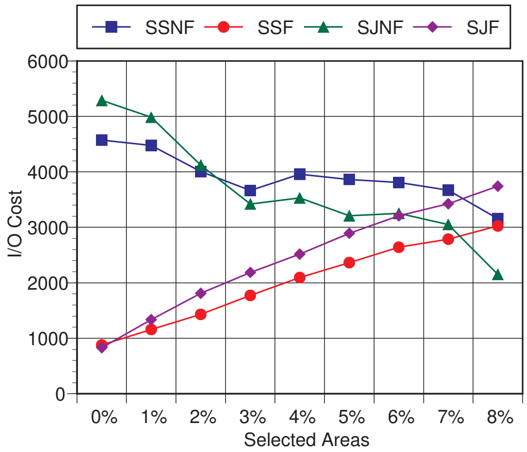

Figure 20: I/O Cost vs. Selected Areas (With Filtering Cost).

the *SSF* and *SJF* increase. The costs of *SSF* and *SJF* are lower than *SSNF* and *SJNF* because the former two strategies have a large part of the computation work done in the filtering process (which is not accounted for in this chart). Including the filtering costs, Figure 20 shows that at around 8% of the *selected areas*[7], the cost of *SSF* catches up with the cost of *SSNF*. While the cost of *SJF* catches up with the cost of *SJNF* at 6% of *selected areas*. Therefore, the filtering strategies, *SSF* and *SJF*, outperform non-filtering strategies, *SSNF* and *SJNF*, for smaller percentages of *selected areas*. However, non-filtering strategies are preferred for bigger percentages of *selected areas*.

5.9 Experiment of I/O Cost vs. the Path Length

As we have seen in previous experiments, the number of expanded links during path finding has an effect on the result I/O cost, but the effect is quite different for each strategy. Note that the number of expanded links depends on the path length of the shortest path, we measure the I/O cost of the four strategies in terms of the path length of the *spatial path query*.

In this experiment, we fix the *buffer size* to 100 and the percentage of *selected areas* to 5% of the total 10,000 areas (Figures 21 and 22). In Figure 21, we do not include the *filtering* costs of the *SSF* and *SJF* strategies, while in Figure 22 we do. We have discussed in previous experiments that the I/O costs of all four strategies directly depend on the number of links expanded by the path finding algorithm. For short path length queries, not much savings can be gained by the filtering processes of the *SSF* and *SJF* strategies because only a smaller number of links are expanded. Increasing the path length of the *spatial path query*, the number of links that need to be expanded during path finding increases dramatically. Since the *SSNF* strategy conducts the spatial overlap operation over the general *R*-tree for each link expansion, its I/O cost thereafter increases dramatically. For

[7] In our experiments, almost no path exists for the percentage of *selected areas* beyond 8%.

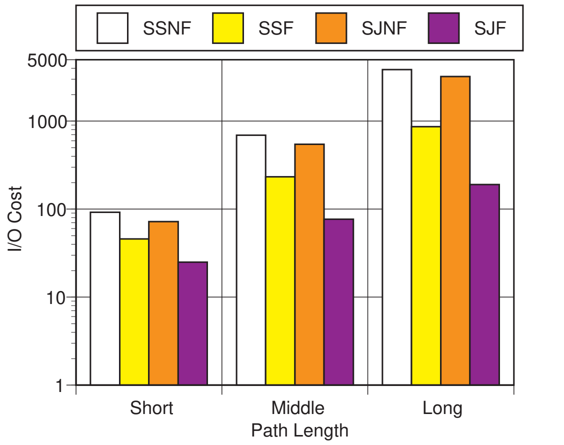

Figure 21: I/O Cost vs. Path Length (Without Filtering Cost).

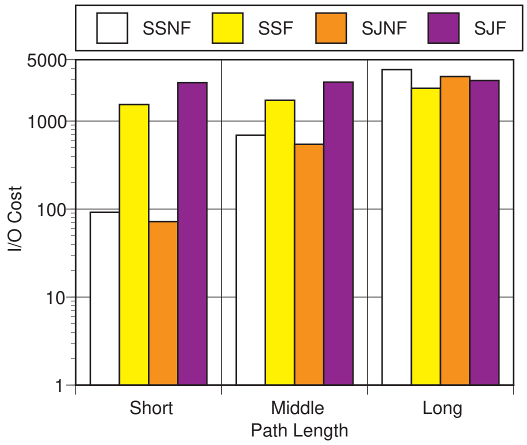

Figure 22: I/O Cost vs. Path Length (With Filtering Cost).

SJNF, although the spatial overlap operation can be simply achieved by a lookup of the general join index, the dramatical increase of the I/O cost is due to the huge size of general join index. The other two filtering strategies, namely *SSF* and *SJF*, have a smaller sized specialized *R*-tree and/or a specialized join index, therefore their I/O costs increase relatively slowly with the increase of path length of the *spatial path query*. Figure 22 depicts the total costs of the four strategies in terms of the path length of the *spatial path query*. It turns out that the filtering processes of the *SSF* and *SJF* strategies impose more I/Os for short and middle length paths. In conclusion, the non-filtering strategies (*SSNF* and *SJNF*) are suitable for short and middle length paths in preprocessing *spatial path queries*, while the filtering strategies (*SSF* and *SJF*) are more appropriate for the long paths.

5.10 Experiment of I/O Cost vs. the Total Number of Areas

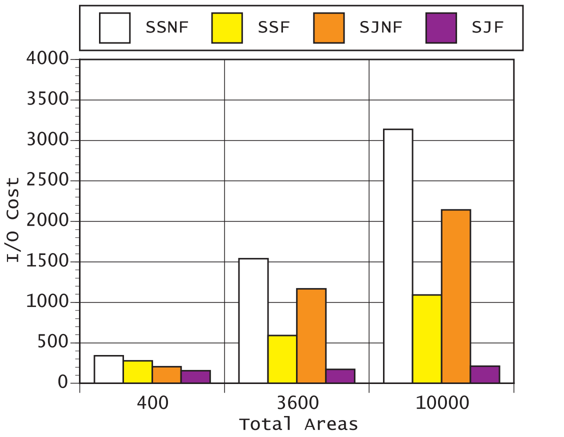

Figure 23: I/O Cost vs. Total Number of Areas (Without Filtering Cost).

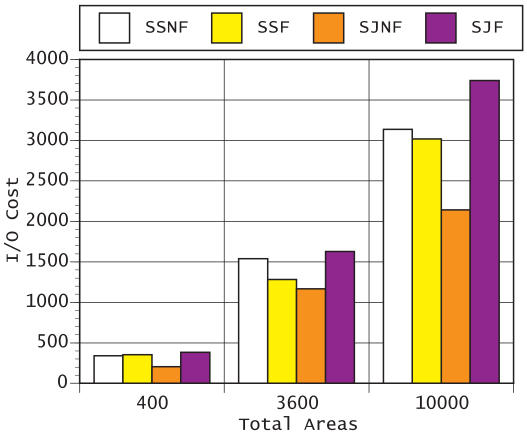

Figure 24: I/O Cost vs. Total Number of Areas (With Filtering Cost).

The total number of areas may also affect the performance of alternative strategies. Thus in this experiment, we generate 400, 3,600, and 10,000 areas of the same size, separately. For each run of the experiment, we assign the *buffer size* to be 100 and choose 8% of the total areas as *selected areas*. The experimental results are depicted in Figure 23 (without filtering cost for *SSF* and *SJF*) and Figure 24 (with filtering cost for *SSF* and *SJF*). It is expected that with the increase of total number of areas, the general *R*-tree and the general join index will increase in size accordingly. This is clearly depicted in Figure 23 where the I/Os for the *SSNF* and *SJNF* increase dramatically when the total number of areas increases. Similarly, the specialized *R*-tree and specialized join index will increase in size with the increase of the total number of areas, which result in the increase of filtering costs for the *SSF* and *SJF* strategies (Figure 24). Figure 24 depicts the overall I/O costs of the four strategies. *SJNF* has better performance over other strategies for all different numbers of area.

6 Related Work

To our knowledge, there has not been any prior research on *spatial path queries* by the database community. Recent database research in path problems focused on pure transitive closure computation without embedded constraints [1, 12]. For single-pair path search, [22] proposed auxiliary structures that contribute to the pruning of the search space, and [2] proposed a clustering scheme based on topological ordering. The path search algorithm we employ in our experiments in this paper is based on spatial clustering optimization [11] that is shown to be better suited for cyclic graphs (GIS maps).

For spatial search, [16] proposed a z-ordering method which maps the multiple-attribute data into single-attribute data. Treating the data space as a grid structure, [15] proposed a Grid File structure capable of accessing multiple-attribute data in a more direct manner. The above access techniques specialize in efficient access to spatial point data and require spatial transformations to search multi-dimensional spatial data relevant to our work. *R*-trees [9] however are designed for such multi-dimensional data. In [3, 21], different variations of the *R*-tree are presented. In our experiments, we deploy an *R*-tree based on [9], but with input data properly packed by their spatial proximity. For spatial join, [4, 8, 13, 14, 18, 19] proposed various techniques. Among them, we use the optimization techniques proposed in [4] to achieve efficient execution of our spatial join. The focus of this paper is not to present new techniques for either path search or spatial queries in isolation, but rather to integrate already existing technologies in both areas to effectively address the combined problem of *spatial path query* processing.

7 Conclusion

In this paper, we address the increasing demand for *customized path queries* in the context of GIS databases by focusing on one class of such queries, namely, *spatial path queries*. Given the lack of prior research on *spatial path query* processing, we study this new class of path queries and propose alternative query optimization decisions categorized along two dimensions. The two dimensions are (1) the timing of the spatial overlap computation between areas and links, and (2) the timing of checking the satisfiability of the spatial query constraint of areas. Based on different alternatives along these two dimensions, we develop four strategies for processing *spatial path queries* in this paper, called *SSNF*, *SSF*, *SJNF*, and *SJF*.

We then propose the implementation approaches for the proposed four strategies using suitable state-of-the-art database technologies. For example, the path search is based on the *Dijkstra* algorithm with spatial clustering optimization [11], the preprocessing approach is implemented using spatial join techniques proposed by [4], and the spatial overlap query is implemented using the *R*-tree index [9] and the spatial intersect algorithm found in [5, 20]. We also outline the step-by-step query processing plan for each strategy.

We present experiment results comparing the proposed strategies under different circumstances controlled by various parameter settings. Based on our experimental results, there is no persistent winner among the four strategies under all circumstances. However we observe: (1) preprocessing spatial queries is advantageous with long paths, smaller buffer sizes, and a smaller percentage of areas relevant to the query constraint even when the preprocessing cost is included, and (2) intermixing path search with spatial queries works better with shorter paths, larger buffer sizes, and a larger percentage of areas relevant to the query constraint. This paper provides an understanding of query processing for complex queries such as *spatial path queries* based on both the cost comparison and experimental evaluation. Such an understanding can be used as a guideline to select the potentially best strategy to process *spatial path queries* under given circumstances. We believe such selective processing is a significant optimization over the naive approach which would be employed in current DBMS systems.

References

[1] Agrawal, R., Dar, S., and Jagadish, H. V., "Direct Transitive Closure Algorithms: Design and Performance Evaluation," *ACM Transactions on Database Systems*, Vol. 15, No. 3, Sep. 1990, pp. 427 – 458.

[2] Agrawal, R. and Kiernan, J., "An Access Structure for Generalized Transitive Closure Queries", *IEEE 9th Int. Conf. on Data Engineering,* 1993, pp. 429 – 438.

[3] Bechmann, N., Kriegel, H., Schneider, R., and Seeger, B., "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of Data,* May 1990, pp. 322 – 332.

[4] Brinkhoff, T., Kriegel, H., and Seeger, B., "Efficient Processing of Spatial Joins Using R-trees", *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data,* 1993, pp. 237 – 246.

[5] Brinkhoff, T., Kriegel, H., Schneider, R., and Seeger, B., "Multi-Step Processing of Spatial Joins", *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data,* 1994, pp. 197 – 208.

[6] Egenhofer, M. J., "What's Special about Spatial? Database Requirements for Vehicle Navigation in Geographic Space", *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data,* 1993, pp. 398 – 402.

[7] Faloutsos, C. and Kamel, I. "Beyond Uniformity and Independence: Analysis of R-tree Using the Concept of Fratal Dimension," *Proc. of the 1994 ACM SIGMOD/PODS Int. Conf. on Principle Of Data,* 1994, pp. 4 – 13.

[8] Gunther, O., "Efficient Computation of Spatial Joins," *Proc. of the 9th Int. Conf. on Data Eng.,* 1993, pp. 50 – 59.

[9] Guttman, A., "R-tree: a dynamic index structure for spatial searching", *Proc. of the 1984 ACM SIGMOD Int. Conf. on Management of Data,* 1984, pp. 45 – 57.

[10] Huang, Y. W., Jing, N. and Rundensteiner, E., "Hierarchical Path Views: A Model Based on Fragmentation and Transportation Road Types," *The Third ACM Workshop on Geographic Information Systems*, Washington, D.C., Nov. 1995.

[11] Huang, Y. W., Jing, N. and Rundensteiner, E., "Effective Graph Clustering for Path Queries in Digital Map Databases", To appear in *Proc. of the Int. Conference on Knowledge Management*, Nov., 1996.

[12] Ioannidis, Y. E., Ramakrishnan, R., and Winger, L., "Transitive Closure Algorithms Based on Graph Traversal," *ACM Transactions on Database Systems*, Vol. 18, No. 3, Sep. 1993, pp. 512 – 576.

[13] Lo, M.L. and Ravishankar, C.V., "Spatial Join Using Seeded Trees," *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data,* May 1994, pp. 209 – 220.

[14] Lo, M.L. and Ravishankar, C.V., "Spatial Hash-Joins," *Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data,* June 1996, pp. 247 – 258.

[15] Nievergelt, J. and Hinterberger, H., "The Grid File: A Adaptable, Symmetric Multikey File Structure", *ACM Transactions on Database Systems*, Vol. 9, No. 1, Mar. 1984, pp. 39 – 71.

[16] Orenstein, J.A. and Merrett, T., "A Class of Data Structures for Associative Searching," *Proc. Symp. on Principles of Database Systems*, 1984, pp. 181 – 190.

[17] Pagel, B., Six, H., Toben, H. and Widmayer, P., "Towards an Analysis of Range Query Performance in Spatial Data Structures," *Proc. of the 1993 ACM SIGMOD/PODS Int. Conf. on Principle Of Data,* 1993, pp. 214 – 221.

[18] Patel, J.M. and DeWitt, D.J., "Partition Based Spatial-Merge Join," *Proc. of the 1996 ACM SIGMOD Int. Conf. on Management of Data,* June 1996, pp. 259 – 270.

[19] Rotem, D., "Spatial Join Indices," *IEEE 7th Int. Conf. on Data Engineering,* 1991, pp. 500 – 509.

[20] Shamos, M.I. and Hoey, D.J., "Geometric Intersection Problems," *Proc. 17th Annual Conf. on Foundations of Computer Science*, 1976, pp. 208 – 215.

[21] Sellis, T., Roussopoulos, N. and Faloutsos, C., "The R^+-Tree: A Dynamic Index for Multi-dimensional Objects," *Proc. of the VLDB Conf.*, Brighton, England, 1987, pp. 3 – 17.

[22] Zhao, J.L. and Zaki, A., "Spatial Data Traversal in Road Map Databases: A Graph Indexing Approach," *CIKM*, 1994, pp. 355 – 362.