CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations

Shashi Shekhar and Duen-Ren Liu

Abstract

Current Spatial Database Management Systems (SDBMS) provide efficient access methods and operators for point and range queries over collections of spatial points, line segments, and polygons. However, it is not clear if existing spatial access methods can efficiently support network computations which traverse line-segments in a spatial network based on connectivity rather than geographic proximity. The expected I/O cost for many network operations can be reduced by maximizing the *Weighted Connectivity Residue Ratio (WCRR)*, i.e., the chance that a pair of connected nodes that are more likely to be accessed together are allocated to a common page of the file. CCAM is an access method for general networks that uses connectivity clustering. CCAM supports the operations of *insert*, *delete*, *create*, and *find* as well as the new operations, *get-A-successor* and *get-successors*, which retrieve one or all successors of a node to facilitate aggregate computations on networks. The nodes of the network are assigned to disk pages via a graph partitioning approach to maximize the WCRR. CCAM includes methods for static clustering, as well as dynamic incremental reclustering, to maintain high WCRR in the face of updates, without incurring high overheads. We also describe possible modifications to improve the WCRR that can be achieved by existing spatial access methods. Experiments with network computations on the Minneapolis road map show that CCAM outperforms existing access methods, even though the proposed modifications also substantially improve the performance of existing spatial access methods.

Keywords: Access Methods, Geographic Information Systems, Network Computations, Spatial Databases, Spatial Networks

1 Introduction

Spatial network databases [19, 29, 43] are the kernel of many important applications, including transportation planning; air traffic control; water, electric and gas utilities; telephone networks; urban management; sewer maintenance, and irrigation canal management. The phenomena of interest for these applications are structured as spatial networks, which consist of a finite collection of the points (i.e. nodes), the line-segments (i.e. edges) connecting the points, the location of the points, and the attributes of the points and line-segments. For example, a spatial network database for transportation applications may store road intersection points and the road segments connecting the intersections. Network computations perform connectivity-based computations including route evaluation, path computation, tour evaluation and location-allocation evaluation [15, 29].

There has been a great deal of research within the database area in the design and evaluation of spatial access methods for point and range queries over collections of points, line-segments, and polygons. Considerable research has also been carried out within the database area in the design and evaluation of algorithms for the shortest path computation. However, there has been little work on the design and evaluation of storage and access methods for network data and for aggregate queries on networks, in which the connectivity relationship is more important than the proximity relationship. Efficient access methods are available for a severely restricted class of networks, namely directed acyclic graphs [5, 10, 20, 28] and directed graphs with limited cycles [4], which do not adequately model many networks of interest, including road-maps.

This paper shows that the expected I/O cost of many network computations can be reduced by maximizing the weighted connectivity residue ratio (WCRR). We propose a connectivity-clustered access method, CCAM, to efficiently support aggregate queries over general networks such as road maps. We use the spatial network data and network computation queries from the domain of Intelligent Vehicle Highway Systems (IVHS) to evaluate the ideas. IVHS is also known as Intelligent Transportation Systems (ITS).

1.1 Example Application : IVHS and Network Analysis

We are particularly interested in transportation applications such as Advanced Traveler Information Systems (ATIS) and Intelligent Vehicle Highway Systems (IVHS). IVHS [1] is currently being developed to improve the safety and efficiency of automobile travel. ATIS is one facet of IVHS which assists travelers with trip planning, navigation perception, analysis and decision-making to improve the convenience, safety and efficiency of travel [8, 39]. An important component of IVHS and ATIS is a spatial network database containing road maps, public transportation routes, and current travel time for segments of the transportation network, which is updated frequently. As shown in Figure 1, ATIS obtains information from different sources, including traffic reports,

scheduled traffic events, sensors and maps, etc. Periodic sensor data might lead to high update rates. The clients of the database include travelers, commuters, drivers on the road, mobile persons with hand-held or portable personal communication devices (PCDs), and users who access information via computers at home, offices, shopping malls, or information centers.

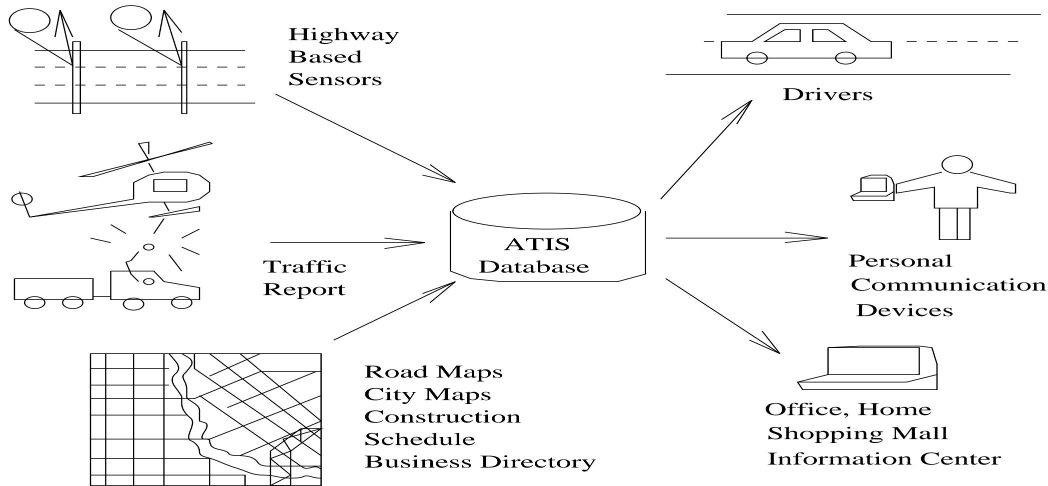


Figure 1: Spatial Network Management in ATIS (Genesis Project at Minnesota)

For ATIS and other applications, an efficient and effective spatial network database is needed to support network analysis [38, 43]. Network analysis represents frequent aggregate queries on spatial networks, such as route evaluation and path computation [43], etc. Route evaluation is concerned with aggregating attribute data over route-units. A route-unit represents a collection of arcs with common characteristics (e.g. name) [29]. *Route evaluation* yields summary information for decision-support applications. For example, utility companies may track the volume of gas/electricity flowing through major pipeline route-units in their networks. Route evaluation for daily commuters often consists of comparing a set of familiar routes based on the current travel-time, congestion, restrictions and other attributes of the transportation networks. Other interesting problems in network analysis include *path computation* [19, 29, 43], which models problems such as shortest path analysis and minimum travel-time route computation, etc.

1.2 Related Work and Our Contributions

We discuss some of the existing access methods, including proximity-based spatial access methods and connectivity-based access methods, which may be adapted for spatial network databases.

Linear-clustering based spatial access methods order the points in multiple dimensions by a space-filling curve, with a specific resolution of the space, and use a one-dimensional access method with this ordering. They

perform transformations on higher-order keys to impose total ordering. Example methods include Z-ordering [32] and Hilbert Curves [2, 11, 22]. Multidimensional B-trees [35] and K-dB-trees [33] establish a correspondence between the levels of the index and dimensions. These approaches limit the opportunities for clustering according to connectivity.

Other spatial access methods capture the isotropic nature of proximity by recursively dividing the space, using a splitting rule to construct a grid or a hierarchy of regions [17]. A survey of these methods can be found in [34]. Some of the representative Isotropic Access Methods (ISM's) include grid files [31], cell-trees [17], R-trees [18] and R^+ trees [36]. Isotropic spatial access methods have traditionally been used to store vector-spatial data such as sets of polygons, and they allow flexible policies which can be adapted to take advantage of connectivity information.

The literature on transitive closure and recursive-query processing has evaluated algorithms for path computations. A survey of the work can be found in [21]. The effect of efficient storage and access methods on the performance of path computations is currently being explored. Most of the proposed methods have looked at storing the nodes of a directed acyclic graph in topological order [28], using a conventional index such as the B-tree. Path computations, such as graph traversal and transitive closure, can be carried out by scanning forward in the file, using a priority queue [28] or a FIFO queue [20]. Topological orders, including depth-first sequence and breadth-first sequence, have been evaluated in [5] for their effectiveness in supporting different graph-traversal problems. Reverse-topological-ordering based methods have also been used to cluster related nodes in the same data page to reduce I/O cost [21]. Finally, the topological ordering method has been extended to graphs with a few cycles in [4].

The methods based on topological order or reverse topological order can be extended to graphs that have many undirected edges as traversals, using the well-known depth-first or breadth-first search strategies. However, methods based on total ordering of nodes are not efficient for general networks, since aggregate queries on networks can no longer be done using a single scan of the data file. Furthermore, none of the proposed access methods takes full advantage of the connectivity properties of a network, due to their reliance on total ordering.

Join-indices [42] can also be used to speed up iterative algorithms for computing transitive closure, and a materialized view can also accelerate path computation. Transitive closure queries can be answered by a look-up in the materialized view. A survey of these techniques can be found in [4]. However, these techniques require a separate structure for each path computation over the same graph and are not space efficient. Meanwhile, static schemes based on the graph-partitioning heuristic, albeit in a different context, were recently used in [41]. The issues involved in dynamic updating effects during insertion and deletion have not been discussed.

Contributions: In the past, most research has focused on the modeling and evaluation of path-computation algorithms. They have provided efficient access structures, based on topological ordering, that support path computations over networks which can be represented as directed graphs with a few cycles. However, little work has been done to design an efficient access method that can support aggregate queries, e.g. route evaluation, over general networks such as road maps, which are strongly connected over the entire graph. Topological ordering-based access methods, when adapted to road maps, do not take advantage of the entire connectivity relationship.

We propose a new access method, CCAM, to efficiently support aggregate queries over general networks such as road maps. CCAM supports the operations of Insert(), Delete(), Create(), and Find() as well as the new operations, Get-A-successor() and Get-successors(), which retrieve one or all successors of a node to facilitate aggregate computations on networks. We adapt a *heuristic graph-partitioning approach* to cluster the nodes of a network into pages based on the connectivity relationship. Ideally, the clustering maximizes the WCRR, i.e., the chances that a pair of connected nodes that are more likely to be accessed together are allocated to a common page of the file. Analysis and experiments show that the proposed method leads to reduced I/O costs and a higher WCRR for many interesting networks.

The literature in the area of graph partitioning [6, 7, 13, 25] has only focused on partitioning static graphs without considering dynamic updates. We address the following two issues. First, the static graph-partitioning approach is not efficient when the entire network cannot fit into main memory. In general, road-maps are very large databases [3, 26], and thus may not fit inside main memory. Second, maintaining a high WCRR in the face of Insert() and Delete() operations, without complete reorganization, is a critical problem. To solve the above two issues, we propose *dynamic reclustering strategies* to handle dynamic updating effects. Alternate heuristic methods are identified and evaluated which maintain a high WCRR without incurring a high reorganization cost during insertion and deletion. An *Incremental Create() operation* is designed to cluster and store networks which cannot fit into main memory. Experiments show that the proposed incremental-create operation is competitive with the static-create operation.

In this paper, we formally describe the CCAM access method by detailing the clustering algorithm, data file and procedures used to implement the operation and dynamic reclustering strategies. We provide algebraic analysis as well as experimental evaluation of CCAM. We focus on a comparative performance study of access methods for network computations over spatial networks. We characterize the structure of network computations over spatial networks to show that maximizing the WCRR reduces the expected cost of many network computations. We describe simple ways of improving the performance of traditional spatial access methods for network computations, based on this fundamental insight. We evaluate representative access methods using spatial network data from the domain of IVHS. The experiments show that the WCRR is an effective predictor of the expected I/O cost of

network computations and the performance of various access methods for network computations. Experiments also show that CCAM outperforms traditional access methods, although their performance is improved significantly by the ideas proposed in this paper.

Outline: Section 2 describes the spatial networks, operations and aggregate queries. We also describe our problem formulation. Section 3 defines the CCAM access method. Section 4 presents an algebraic analysis. Section 5 describes the experiment design, and Section 6 presents the experimental observations and results. Finally, Section 7 summarizes our conclusions and suggests future work.

2 Basic Concepts

2.1 Spatial Networks, Operations and Aggregate Queries

A spatial network is a special kind of graph, with nodes located in a two-dimensional or three-dimensional euclidean space. Unlike raster and vector data, spatial network data is characterized by rich connectivity. A spatial network G = (N, E) consists of a node set N and an edge set E. Each element u in N is associated with a pair of real numbers (x,y) representing the spatial location of the node in an euclidean plane. Edge set E is a subset of the cross product N*N. Each element (u, v) in E is an edge that joins node u to node v. There are attributes associated with the nodes and edges. In general, spatial networks can be represented in many different ways. We will focus on the adjacency-list oriented representation, which has been used quite frequently in database research [23]. In this representation, a spatial network is modeled as a list of nodes, and each node has properties including the successor-list and predecessor-list, which represent the outgoing and incoming edges. The predecessor-list facilitates updating the successor-lists during the insertion and deletion of nodes.

Both aggregate queries on networks and the management of network data require that the following set of operations be efficiently supported. Detailed definitions of these operations are given in Section 3.

1. Create: <list of node records> → Network
2. Find: <node-id, Network> → node properties
3. Insert: <node-id, node-properties, Network> → Network
   Insert: <edge, edge-properties, Network> → Network
4. Delete: <node-id, Network> → Network
   Delete: <edge, edge-properties, Network> → Network

5. Get-successors: <node-id, Network> → list of <node-id, node-properties> of successors

6. Get-A-successor: <node-id, successor-id, Network> → node-properties of the successor

The first four operations are common to data types other than aggregate queries on networks. Unlike point and range queries, network computations access data by connectivity and by traversal order. Network computations use topological operations such as Get-successors() and aggregate sequence operations such as Find() and Get-A-successor().

The Get-A-successor() and Get-successors() operations are unique to aggregate queries on networks, and they retrieve one or all successors of a node. **Get-A-successor()** retrieves a specified successor of a given node. **Get-successors()** retrieves the records for all successor nodes of a given node. For example, Get-A-successor() is used in route evaluation queries, while Get-successors() is used in graph search algorithms like $A^{\star}$ [38]. While Get-successors() and Get-A-successor() can be implemented as a sequence of Find() on relevant successors, more efficient implementations are possible by defining that operation as distinct. The Get-successors() and Get-A-successor() operations represent the dominant I/O cost of many aggregate queries on networks [19, 23, 28, 38], including route evaluation and path computations.

Route Evaluation

To derive aggregate properties, route evaluation queries over route-units in networks may require the retrieval of all nodes and all edges in the specified route-units. A route specifies a sequence of nodes $n_1, n_2, \ldots, n_k$ and edges $< n_1, n_2 >, < n_2, n_3 >, \ldots, < n_{k-1}, n_k >$. An aggregate property of a route is a function of the properties of the nodes and edges in the route. An aggregate property of a route can be computed by a sequence of Find() operations on relevant nodes and edges. Alternatively, it can be processed as a sequence of Get-A-successor() operations, e.g. Find(n_1), Get-A-successor(n_1, n_2), ..., Get-A-successor(n_{k-1}, n_k). Thus, the efficient implementation of Get-A-successor() operations reduces the total I/O cost for route evaluation queries.

Path Computations

Search algorithms for path computations such as the breadth-first search, depth-first search, $A^{\star}$ and Dijkatra's consist of iterations. Each iteration is usually centered around a node called the current node for the iteration. Computations in each iteration often access the nodes on the successor-list via the Get-successors() operation. The quantitative models for the I/O cost of several path computations are summarized in [30]. These models are discussed in detail and validated in [37]. These models show that efficient implementation of the Get-successors() operation leads to reduced I/O cost for many path computations.

2.2 Problem Formulation

Given network operations, including Get-A-successor() and Get-successors(), our goal is to find storage and access methods which can provide efficient support for frequent network operations in terms of expected I/O cost.

Theorem 1 *The expected cost of network operations (e.g. Get-A-successor()) is minimized by maximizing the* Weighted Connectivity Residue Ratio (WCRR)*, where*

$$WCRR = \frac{Sum\ of\ w(u,v)\ such\ that\ Page(u)\ =\ Page(v)}{Total\ sum\ of\ weights\ over\ all\ edges}.$$

Proof: *See Section 4.1.* □

The weight $w(u,v)$ associated with $edge(u,v)$ represents the relative frequency of a query accessing nodes u and v together. Intuitively, maximizing the WCRR maximizes the chances that a pair of connected nodes that are more likely to be accessed together are allocated to a common page of the file. The expected cost of Get-A-successor() is predicted by the WCRR. The WCRR also effectively predicts the cost of Get-successors() and Delete() as shown in Section 6.1, even though additional parameters (e.g. correlation of successors' locations) can affect performance. The main effect of the access method on the I/O cost of many aggregate queries can thus be predicted from the WCRR. A higher WCRR indicates lower I/O cost for aggregate queries on networks.

Theorem 1 suggests that the expected cost of network operations and aggregate queries over a network is reduced by designing an access method customized to maximize the WCRR or the sum of the weights over the unsplit edges. It can easily be shown that the problem of partitioning the nodes of a network into pages of a given size, so as to maximize the WCRR, is an instance of the graph-partitioning problem defined in [25]. The graph-partitioning problem is to partition the nodes of a graph with costs on its edges into subsets of given sizes, so as to minimize the sum of the costs on all the cut edges. Although the graph-partitioning problem is NP-complete [14], many good heuristics based on spectral partitioning [6] and iterative approaches [7, 13, 25] have been proposed to solve this problem efficiently. The implementation of CCAM operations takes advantage of these heuristics.

The WCRR model is proposed on the basis of the available database statistics on access frequencies. One way to gather such statistics would be to record the frequency of query occurrence and the access frequencies of nodes and edges. Another source of such statistics is the application domain. For example, in transportation, information about the capacity and use of different road-segments (edges) is often available for major roads. If database statistical information is not available, we can still use the network topology to develop a simplified model, the *Connectivity Residue Ratio (CRR)*, which is a special case of the WCRR model which assumes that

each edge in the network is equally likely to be accessed by aggregate queries over the network.

$$CRR = \frac{Total\ number\ of\ unsplit\ edges}{Total\ number\ of\ edges}.$$

An unsplit edge (u,v) is characterized by page(u) = page(v).

3 CCAM: Connectivity-Clustered Access Method

CCAM clusters the nodes of the network via graph partitioning, using the ratio-cut heuristic described in appendix A. Other graph-partitioning methods can also be used as the basis of our scheme. In addition, an auxiliary secondary index is used to support the Find(), Get-A-successor() and Get-successors() operations. The choice of a secondary index can be tailored to the application. We use the B^+ tree with Z-order [32] in our experiments, since the benchmark networks are embedded in geographical space. Other access methods such as the R-tree [18] and Grid File [31], etc. can alternatively be created on top of the data file, as secondary indices in CCAM to suit the application. In this section, we describe the file-structure and procedures used to implement the various operations on networks.

3.1 Connectivity-Clustered Data File

For each node, a record stores the node data, coordinates, successor-list and predecessor-list. A successor-list (predecessor-list) contains a set of outgoing (incoming) edges, each represented by the node-id of its end (start) node and the associated edge cost. The successor-list is also called the adjacency-list, and is used in network computations. The predecessor-list is used in updating the successor-list during the Insert() and Delete() operations. We will refer to the neighbor-list of a node x as the set of nodes whose node-id appears in the successor-list or predecessor-list of x. We note that the records do not have fixed formats, since the size of the successor-list and predecessor-list varies across nodes.

In contrast with the previous topological ordering based approach [28], CCAM assigns nodes to the data page by a graph partitioning approach, which tries to maximize the WCRR. Each data page is kept at least half full whenever possible. Records of the data file are not physically ordered by node-id values. A primary index cannot be created without renaming the nodes to encode disk-page information in the node-id, and it requires additional overhead during update operations. Therefore, a secondary index is created on top of the data file, and an index entry is created for each record in the data file.

Since our benchmark networks are embedded in geographic space, (x, y) coordinates for each node are also stored in the record. A B^+ tree with Z-ordering [32] of the (x, y) coordinates is used to order the secondary

index. It can support point and range queries on spatial databases. The Z-order of a coordinate x, y is computed by interleaving the bits in the binary representation of the two values.

Example: In Figure 2, a sample network and its CCAM is shown. The left half of Figure 2 shows a spatial network. Nodes are annotated with the node-id (an integer) and geographical coordinates (a pair of integers). To simplify the example, the node-id is an integer representing the Z-order of the (x, y) coordinates. For example, the node with the coordinates (1, 1) gets a node-id of 3. The solid lines that connect the nodes represent edges. The dashed lines show the cuts and partitioning of the spatial network into data pages. There exists a cut on edge e(u, v) if node u and node v fall into different partitions. The partitions are (0, 1, 4, 5), (2, 3, 8, 9), (6, 7, 12, 13) and (10, 11, 14, 15). The right half of Figure 2 shows the data pages and the secondary index. We note that the nodes are clustered into data pages by CCAM, using a graph-partitioning approach. Nodes in the same partition set are stored on the same data page. They are not physically ordered by their node-id values. A secondary index ordered by node-id is used to facilitate the Find() operation.

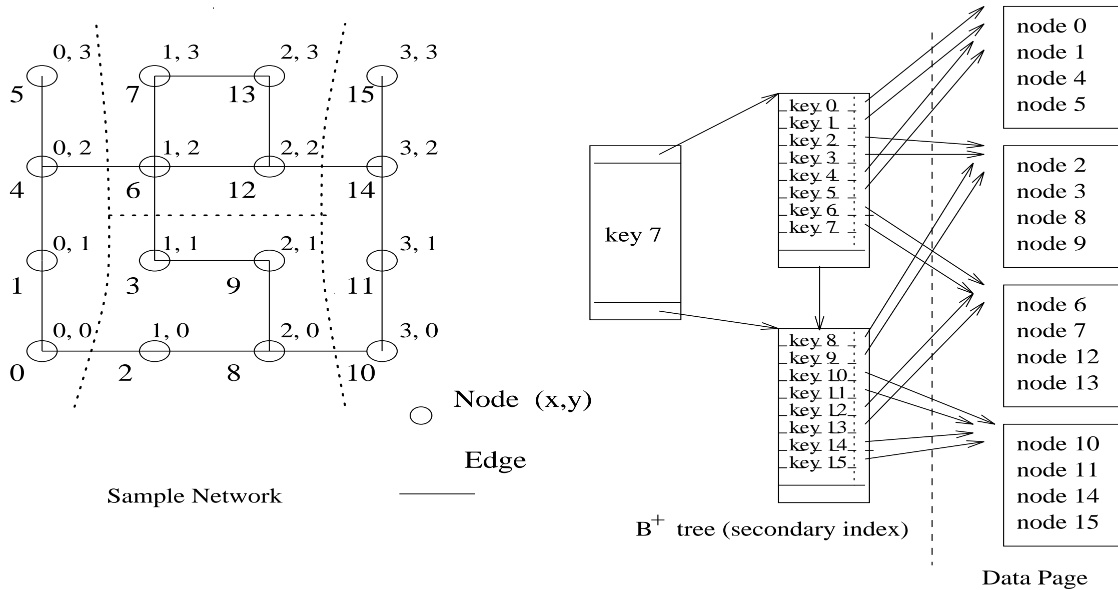


Figure 2: Clustering and storing a sample network (key represents spatial order)

3.2 Create(): Creation of CCAM

The Static-Create() algorithm is based on a graph partitioning approach. First, the nodes of the network are clustered via the cluster-nodes-into-pages() algorithm, which returns a set of pages. Second, the nodes (records) which belong to the same subset are stored on the same data page, and an index entry for each node is created and inserted into the B^+ tree, based on the node-id values which represent the Z-order of the location of the

nodes in space.

Figure 3 shows the connectivity-based clustering algorithm for top-down clustering using the 2-way-partition algorithm. Each subset contains at least min-page-size bytes. We repeatedly apply the 2-way-partition() to cluster the graph. After applying the 2-way-partition() algorithm, two subsets return. We keep on applying the 2-way-partition() algorithm to the subset which exceeds the page-size, until all subset sizes are less than the page-size. Notice that sizeof(A) = $\sum_i$ size of record(i), node i $\in$ A.

```
Procedure: cluster-nodes-into-pages (V: set of nodes;
        E: set of edges; page-size): return set of pages;
 F,P : set of page (partition) of nodes;
 V', A, A' : set of nodes;
begin
 Initialize F = {V}; P = {}; MinPgSize = ⌈ page-size/2 ⌉;
 while F is not empty do
    Choose a V' ∈ F, E'={(u,v)|(u,v)∈E, u∈V' and v∈V'};
    Remove V' form F;
    <A, A'> = 2-way-partition(V', E', MinPgSize);
 // comment: sizeof(A) > MinPgSize; sizeof(A') > MinPgSize;
    if sizeof(A) > page-size then add A to F else add A to P;
    if sizeof(A') > page-size then add A' to F else add A' to P;
 endwhile
 return P;
end;
```

Figure 3: The Connectivity Clustering Algorithm

We adapt Cheng and Wei's two-way ratio-cut heuristic algorithm [7] which is described in appendix A, as the basis for implementing the 2-way-partition() algorithm. The 2-way-partition() algorithm partitions a given set into two subsets by trying to minimize the total weight on the edges in the cut-set, i.e., maximizing the WCRR. The 2-way-partition algorithm [7] adapts the iterative approach, which starts from an initial partition (i.e. two subsets), and then iteratively moves nodes across subsets in an attempt to achieve a global minimum weight on the edges in the cut set. At each pass, the algorithm iteratively selects an unlocked node from two subsets with the largest ratio gain, moves the node to the other side, and locks it, until all the nodes are locked. The process repeats until no further accumulated positive gain is possible. The implementation is based on the bucket-list data structure [13] and requires a time complexity of O(|E|) with respect to the number of edges |E|.

Other graph-partitioning methods can also be used as the basis of our scheme. In fact, M-way partitioning [25, 45] may be used to further improve the result of partitioning, if computation complexity and CPU cost is not a concern.

The Incremental Create() Operation

The Static-Create() operation is not efficient when the entire network does not fit inside main memory. The Incremental Create() operation is designed to handle very large networks. The incremental Create() operation is implemented as a sequence of Add-node() operations, which are similar to the Insert() operations described in Section 3.4.1. The Add-node() operation does not need to update the successor and predecessor lists, since the node records initially presented to create a file can be pre-processed to have the proper values for the predecessor-list and successor-list. This operation will, however, use incremental clustering and reorganization to improve the WCRR, as discussed in Section 3.4. We use **CCAM-D** to denote the implementation of Incremental Create() as a sequence of Add-node() operations. **CCAM-S** denotes the implementation of Static Create().

Incremental Create() is different from bulk loading [44]. Incremental Create() focuses on loading the entire network, which occurs the most frequently in application domains such as transportation. An analysis of appending a partial network to an existing network is outside the scope of this paper.

3.3 Efficient Support of Search Operations

Find(): Retrieve the record of a given node-id

Using the given node-id value, we can retrieve the desired record from the disk by searching the secondary index to read the appropriate data page. Once the appropriate disk block is transferred to the main memory buffer, a search can be carried out for the desired record within the data page.

Get-A-successor(): Retrieve a specified successor of a given node

In principle, the buffered data-page containing the given node is likely to contain the specified successor node if the WCRR is high. Thus the buffered data-page should be searched first. If the desired successor node is not in the buffer, then a Find() operation is needed to retrieve it.

Get-successors(): Retrieve records for the successor nodes of a given node

In principle, when a data page is fetched for the purpose of retrieving the current node (i.e., the given node), all successor neighbors stored in the same data page as the current node would be accessed without further I/O. Node-id values of successor nodes can be extracted from the set of successor-lists stored in record(x). Then, records for neighbors can be retrieved by searching the buffer in the main memory first. Since CCAM clusters nodes in trying to maximize the WCRR, there is a high probability that many successors will be located in the same disk page as node x. This implies that successors are very likely to be found by searching the main memory buffer. Otherwise, a Find() operation is performed to retrieve the records of successors not in that page of the node. The Get-successors() procedure can be improved further by checking all the pages brought into the main

memory buffers by the Find() operation, to determine whether additional neighbor records can be extracted without additional Find() operations. We note that adequate buffering of these pages may perform part of this optimization in some cases; for example, when the number of available buffers is greater than the number of successors of node x.

3.4 Maintenance and Dynamic Reclustering Strategies

There are two basic maintenance operations: Insert() and Delete(). Each can take an argument of an edge or a node. These operations change connectivity relationships, and may make the existing partitioning of the network into pages obsolete. Local reorganizations of the data pages may be needed to improve the WCRR. Intuitively, the data sets chosen for reorganization should be those data pages which are related via the connection between nodes. We adopt the notion of the page access graph (PAG) [27] to formalize the connectivity relationship between data pages.

Definition 1 (Page Access Graph) *Let $G = (V, E)$ be the given network. P is called a page of G if and only if P is a set of records, such that for each record(x) $\in P$, $x \in V$ and all records $\in P$ are stored in the same disk data page, i.e., the total size of the records included in P is at most full disk page size. Let each of $P_1, P_2, \cdots, P_n$ be a page of G. Then the page access graph (PAG) $G_p = (V_p, E_p)$, where V_p is a set of pages and E_p is a set of edges, defined as follows:*
$V_p = \{P_1, P_2, \cdots, P_n\}$,
$E_p = \{(P_i, P_j) \mid \exists\, x, y \text{ such that } x \in V, y \in V, (x, y) \in E, record(x) \in P_i, \text{ and } record(y) \in P_j\}$

Definition 2 *:*
· *Is-Neighbor-Page(P,Q) = true iff either (P,Q) $\in E_p$ or (Q,P) $\in E_p$.*
· *NbrPages(P$\in V_p$) = $\{$Q $\mid$ Q$\in V_p$ and Is-Neighbor-Page(P,Q)$\}$.*
· *Page($x \in V$) = Q, where Q$\in V_p$ and record(x)$\in$ Q.*
· *PagesOfNbrs(x$\in V$) = $\{Page(u) \mid u \in succ(x) \cup pred(x)\}$.*

The principle of our dynamic reclustering strategy is to reorganize a suitable set of pages which are connected in the page access graph. The reorganization is performed by applying the cluster-nodes-into-pages() algorithm described in Figure 3 to recluster the subnetwork formed from the nodes in the set of pages to be reorganized. For such a set of pages to be reorganized, the choice might be based not only on maximizing the WCRR, but also on reducing the overhead required for reorganization.

The key issue in the design of dynamic reclustering strategies is to identify a reorganization policy which yields a high WCRR without incurring high I/O costs. The reorganization policies can be defined in terms of the concept of a page access graph, as shown in Table 1. The table identifies a set of pages to be reorganized, given the argument type (edge or node) and the reorganization policy (1st, 2nd, higher). It assumes that Page(x)

Reorganization Policy	Set of Pages to be Reorganized		
	argument = edge(u,v)	**argument = node x**	**Guiding Principle**
First order	none handle underflow/overflow	none handle underflow/overflow	avoid or delay reorganization
Second order	{Page(u), Page(v)}	{Page(x)} ∪ PagesOfNbrs(x)	reorganize pages which must be updated anyhow
Higher order	1. NbrPages(Page(u))∪{Page(u)} ∪NbrPages(Page(v))∪{Page(v)} or 2. {Page(u)}∪PagesOfNbrs(u)∪ {Page(v)}∪PagesOfNbrs(v) or 3. all pages in data file	1. {Page(x)} ∪ PagesOfNbrs(x) ∪ NbrPages(Page(x)) or 2. all pages in data file	reorganize more pages than second order policy
Page(x) = page selected to place x in Insert() or page containing x in Delete()			

Table 1: Set of Pages reorganized by different Policies for Maintenance

represents the page containing x, or selected to contain x in the event of Insert(x). To simplify this table, overflow and underflow events are abstracted and are discussed separately. The second order policies are designed to avoid additional I/O overhead in reorganization. Second order and higher order policies can incur a high CPU cost if the average degree of nodes increases. Other reorganization policies can be built around the basic policies shown in Table 1. For example, a lazy or delayed reorganization policy may reorganize NbrPages(P) after a certain number of updates to page P.

The efficient implementation of the first-order and second-order policies is linked to the buffering of the pages retrieved during Get-successors(). Thus, connectivity-based clustering in CCAM is suited to the first and second order policies. The efficient implementation of the higher order policies may require additional data structures. NbrPages(P$\in V_p$) can be retrieved efficiently if the page access graph is materialized to avoid repeated traversal of the secondary index. We choose not to materialize the page access graph, since it requires additional redundant data structures.

Choice of Reorganization Policy

The order of reorganization policy represents the order of overhead required during the update. In general, a higher order policy can yield a higher WCRR, but it incurs higher overhead. Let the data reorganization cost be the time overhead spent in reorganizing the data pages, and let the data retrieval cost be the time spent in searching operations and aggregate queries. By choosing a proper policy, the total cost of data reorganization should be kept below the saving of data retrieval. In the rest of this paper, the higher-order policy is represented by its first reorganization example, as listed in Table 1.

3.4.1 Insert(): Insert a new node or edge

The Insert() operation is used to add a new edge or a node to the data file. The insertion of an edge(u,v) is allowed only if nodes u and v exist in the data file. The new edge requires updating the successor-list for u and the predecessor-list for v, which can be accomplished by retrieving the relevant pages via index-traversal and then updating these pages. Reorganization may be carried out on the pages specified by Table 1, via the cluster-nodes-into-pages() procedure described in Figure 3.

By the insertion of a node, we mean the insertion of a node-record, which contains node properties such as the adjacency-list (successor-list and predecessor-list), other attributes of the node, and the attributes of the edges connected to the node. During the insertion of a new node x, a data page must be selected in which to store the new node. To maximize the WCRR, the new node should be placed in a page containing many neighbors connected via edges having higher weights. Page selection may be accomplished by ranking the pages by the total weight on the edges to the neighbors of x located in the page, to choose the page with the maximum weight on edges to the neighboring nodes of x which also has space to accommodate x. The successor-list of the predecessors of x, as well as the predecessor-list of the successors of x, should be updated to complete the operation. The pages containing the successors and the predecessors of x can be retrieved by using the secondary index for these updates. In the case of overflow in any of the updated pages, the overflow page is split into two pages, via the cluster-nodes-into-pages() procedure. Reorganization may be carried out on the pages specified by Table 1, via the cluster-node-into-pages() procedure described in Figure 3. For example, the second order policy will reorganize the set of pages described by $\{\text{Page(x)}\} \cup \text{PagesOfNbrs(x)}$ as per Table 1. Finally, the index is updated to reflect the changes to the data file. Figure 4 shows a procedural description of Insert() for node arguments.

3.4.2 Delete(): Delete a node or edge

The Delete() operation can be used to delete an edge or a node from the data file. The deletion of an edge (u,v) is accomplished by updating the successor-list of u and the predecessor-list of v, and by accessing Page(u) and Page(v) via the secondary index. In the case of underflow, data-page merging may be required. In addition, reorganization may take place according to the specified policy.

The deletion of a node is implemented in a similar way. Figure 5 shows the delete algorithm. The data page P that stores the record(x) to be deleted can be retrieved by using the node-id value of node x. If the deletion makes the page underflow, two data pages might be merged to increase data-page utilization. We can simply choose a neighboring page Q of P from PagesOfNbrs(x) to be merged with P. If Q and P cannot be merged into one page, they are distributed between the two pages, using the cluster-nodes-into-pages() procedure. The selection of page

```
Procedure: Insert (x: node-id; record(x): node-properties;
                policy: reorganization-policy)
begin
 retrieve PagesOfNbrs(x);
 if PagesOfNbrs(x) is empty then
     insert record(x) into an available disk page P;
     insert index entry (node-id x, disk address of P);
 Otherwise,
     update succ-list and pred-list of neighbors(x);
     select a page P from PagesOfNbrs(x) to put record(x);
     if (policy == first-order policy) then
         for each page Q in PagesOfNbrs(x) do
             if Q overflow then split Q into two pages
             else if Q has been modified then Write Q;
     else
// comment : local reorganization of few pages connected to Page(x)
         Reorganize(x, policy);
end;
```

Figure 4: The Insert Algorithm for Nodes (records)

Q may be accomplished by ranking the pages by the total weight on the edges that cross page P and the number of data-bytes in the page. Since the connectivity relationship is then changed, data reorganization might be used to further increase the WCRR.

4 Analytical Evaluation and Cost Models

4.1 Is the WCRR the Right Metric?

In this subsection, we prove theorem 1, that the expected cost of network operations (e.g. Get-A-successor()) is minimized by maximizing the WCRR, as stated in Section 2.2. We restate the theorem for the convenience of readers.

Theorem 1 *The expected cost of network operations (e.g. Get-A-successor()) is minimized by maximizing the* Weighted Connectivity Residue Ratio (WCRR).

Proof : Given a graph $G = (N, E)$ and the edge cut-set E_C, let an *unsplit* edge (u, v) be characterized by page(u) = page(v). Let the unsplit edge set denoted by E_R be $E - E_C$. The cost of accessing the pair of nodes

```
Procedure: Delete (x: node-id; policy: reorganize-policy)
begin
  retrieve P = Page(x); retrieve PagesOfNbrs(x);
  update succ-list and pred-list of neighbors(x);
  delete x from P; delete index entry of x;
  if (policy == first-order) then
      if page P underflow then
          select a page Q from PagesOfNbrs(x);
          perform data page merging on {P, Q};
      for each page Q in {PagesOfNbrs(x), P} do
          if Q has been modified then Write Q;
  else
      Reorganize(x, policy);
end;
```

Figure 5: The Delete Algorithm for Nodes

connected by edge $(u, v) \in E$, $c(u, v)$, is defined by

$$c(u,v) = \begin{cases} \sigma & \text{if } page(u) = page(v), \\ \tau & \text{if } page(u) \neq page(v), \ \ \tau \geq \sigma. \end{cases} \tag{1}$$

Let the weight on edge(u,v), denoted by $w(u, v)$, represent the relative frequency of network operations that access the pair of nodes connected by edge(u,v). Let $g(u, v)$ be equal to $\frac{w(u,v)}{\sum_{(u,v) \in E} w(u,v)}$. Then $g(u, v)$ is the Probability[pair of nodes connected by edge(u,v) used in network operations $\mid (u, v) \in E$]. It is clear that $\sigma \leq \tau$ and $\sum_{(u,v) \in E} g(u, v) = 1$. The expected cost of a network operation per edge, denoted by Θ, is equal to $\sum_{(u,v) \in E} c(u, v) \cdot g(u, v)$. We can derive the following:

$$\Theta = \sum_{(u,v) \in E_C} c(u,v) \cdot g(u,v) + \sum_{(u,v) \in E_R} c(u,v) \cdot g(u,v) = \tau \cdot \sum_{(u,v) \in E} g(u,v) - (\tau - \sigma) \cdot \sum_{(u,v) \in E_R} g(u,v)$$

From the fact that $\sum_{(u,v) \in E} g(u, v) = 1$, we can further derive the following equation:

$$\Theta = \tau - (\tau - \sigma) \cdot \sum_{(u,v) \in E_R} g(u,v) \tag{2}$$

In other words,

$$\Theta = \tau - (\tau - \sigma) \cdot WCRR, \ \ where \ WCRR = \sum_{(u,v) \in E_R} g(u,v) = \frac{\sum_{(u,v) \in E_R} w(u,v)}{\sum_{(u,v) \in E} w(u,v)} \tag{3}$$

Thus, maximizing the WCRR minimizes Θ, the expected cost of accessing an edge. In the case that each edge is equally likely to be accessed by network operations, i.e., $g(u, v)$ is uniform distribution, $g(u, v) = \frac{1}{|E|}$ for any edge $(u, v) \in E$, then this implies that

$$\Theta = \tau - (\tau - \sigma) \cdot CRR, \ \ where \ CRR = \sum_{(u,v) \in E_R} g(u,v) = \frac{\mid E_R \mid}{\mid E \mid}$$

□

The theorem is exact for the Get-A-successor() operation. The I/O cost of other network operations is not totally determined by WCRR. However, WCRR is a good predictor of their costs, as shown in Section 4.2.

4.2 Cost Modeling Framework for Network Operations

In this section, we provide simple algebraic cost models for the I/O cost of network operations, using the CRR measure of access methods. For simplicity, we assume that each edge is equally likely to be accessed by network operations. This assumption is made only to simplify the analysis. However, our techniques can also take advantage of non-uniform weights, if the statistics are available for the road segments chosen for network operations. The experimental evaluation considers the general case, where weight distribution is not uniform.

Table 2 lists the symbols used to develop our cost formulas. λ denotes the average number of nodes in the neighbor list of a node. The neighbor list of a node x includes all the neighbors of node x, while the successor (adjacency) list of a node x only contains the successor neighbors of node x.

Symbol	Meaning
$\vert A\vert$	Average number of nodes in the successor-list of a node
α	CRR = Pr.[Page(i)=Page(j)] for edge(i, j)
λ	Average number of nodes in the neighbor-list of a node
γ	Average blocking factor
L	Number of nodes in aggregate queries over routes

Table 2: Symbols used in Cost Analysis

Cost Modeling for Search Operations

The algebraic cost of search operations is listed in Table 3. We list the number of data pages accessed for each operation. The Find() operation needs at most one data page access. The Get-successors() operation retrieves all the successors of a given node x. We assume that the data page containing node x is located in the main memory. On the average, $\alpha * |A|$ successors are in the same page as x (assuming $\gamma > \alpha * |A|$), and can be processed first to reduce the need for additional I/O, even if there is only one buffer. Additional data page accesses are needed to retrieve the other $(1 - \alpha) * |A|$ successors, and it takes at most $(1 - \alpha) * |A|$ data page accesses. Thus the expected cost is $(1 - \alpha) * |A|$. Similarly, the Get-A-successor() operation needs $(1 - \alpha)$ data page accesses on the average to retrieve the successor node of a given node x, assuming that the data page containing node x is located in the main memory. In general, route evaluation queries can be modeled as a sequence of Get-A-successor() operations. Then the number of data page accesses for route evaluation queries over L nodes is approximately $1 + (L - 1) * (1 - \alpha)$, assuming buffering with one data page and no global query optimization.

Operation	Data Page Accesses
Find()	**1**
Get-successors()	$(1-\alpha) * \lvert A \rvert$
Get-A-successor()	$1-\alpha$
Route Evaluation	$1+(L-1)*(1-\alpha)$

Table 3: Cost Analysis for Retrieval Operations

Cost Modeling for Update Operations

We analyze the number of data pages accessed in each update operation. Table 4 summarizes the worst case retrieval (*Read*) cost for Insert() and Delete() operations, under different reorganization policies. In general, the *Write* cost is equal to the *Read* cost, unless there is underflow or overflow. To simplify our comparison, we assume these costs are the same. A detailed analysis is provided in appendix B.

Policies	**Data Page Accesses**	
	Insert()	**Delete()**
first-order	λ	$1+\lambda*(1-\alpha)$
second-order	λ	$1+\lambda*(1-\alpha)$
higher-order	$\lambda+\gamma*\lambda*(1-\alpha)$	$\gamma*\lambda*(1-\alpha)$

Table 4: Simplified worst case retrieval cost for update operations

The cost modeling analysis for network operations shows that the efficiency of the Get-A-successor(), Get-successors() and Delete() operations depends on parameter α, i.e., the CRR. With a higher CRR, the cost of these operations is lower. CCAM clusters nodes of networks via a graph partitioning approach, and thus can achieve a higher CRR than the other methods. It is interesting to note that the cost of the Insert() operation cannot be predicted from the CRR, since the model cannot capture the clustering efficiencies for the neighbors of a new node being inserted.

4.3 Theoretical Comparison of CCAM with Other Methods

In this section, we compare the access methods for spatial networks using the algebraic cost models of network operations. Our intention is to characterize the effect of secondary index used with CCAM, since many competitors can use a non-dense primary index. The primary indices on node-id differ in depth due to different page-formats and branching factors. The Grid-File [31] provides a fixed depth. A B^+ tree is likely to have a higher branching factor and lesser depth than the Cell-tree [17] for a fixed page-size and a given data set. CCAM can use any index type (e.g. B^+-tree, Grid File, Cell-tree) as a secondary index. The depth of a chosen secondary-index-type (e.g. B^+-tree) is likely to be slightly more than the corresponding non-dense primary-index-type (e.g. B^+-tree with DFS-order) for the same data set and page-size. In this section, we illustrate the comparison methodology

by comparing CCAM using B^+-tree secondary index with the DFS-ordered B^+-tree primary index (DFS-AM) and the Grid File primary index. The methodology can be used to compare other combinations such as CCAM with B^+-tree secondary index, CCAM with Cell-tree secondary index, Cell-tree primary index, etc. The general conclusions would be very similar to those found in the illustration.

We choose the Grid File [31] and DFS-AM [5, 28] as representative of spatial access methods and connectivity-based access methods, respectively, to compare with CCAM. DFS-AM is the extension of topological-ordering based files to general graphs. DFS-AM orders the nodes by a depth-first traversal, and DFS-AM uses a primary index on the ordered node-id that is generated by the traversals. The expected worst case retrieval costs of various network operations for alternative access methods are shown in Table 5. We note that the cost of Insert() and Delete() for CCAM represents the cost of CCAM that uses the first/second reorganization policy. Z_C, Z_T and Z_G represent the cost of accessing a node (record) in CCAM, topologically-ordered files (DFS-AM) [28] and the Grid Files, respectively. The entries listed in Table 5 are derived by adding the cost of index traversal to the cost of network operations, as discussed in Section 4.2, Tables 3 and 4. For example, Get-A-successor() needs (1-α) data pages and $(Z_C - 1) * (1 - \alpha_C)$ index pages, i.e. $Z_C(1 - \alpha_C)$ pages in the absence of prior buffering.

The cost model makes several assumptions to simplify the discussion. It accurately models the cost of data pages accessed and index pages accessed, assuming that only the root node of the index tree is initially in the buffers. It also assumes that the same number of index pages are retrieved for each data page accessed. Finally, for simplicity, the model assumes that the total I/O cost is proportional to the *Read* cost. Many of these assumptions can be set aside to derive an accurate and detailed model in future work. However, the simplified model is adequate for our discussion about the relative roles of the CRR and the indices in determining the total I/O cost.

	I/O cost for retrieving index pages and data pages		
Operation	**CCAM**	**DFS-AM**	**Grid File**
Find()	Z_C	Z_T	Z_G
Get-A-successor()	$Z_C(1-\alpha_C)$	$Z_T(1-\alpha_T)$	$Z_G(1-\alpha_G)$
Get-successors()	$Z_C((1-\alpha_C)\|A\|)$	$Z_T((1-\alpha_T)\|A\|)$	$Z_G((1-\alpha_G)\|A\|)$
Insert()	$Z_C * \lambda$	$Z_T * \lambda$	$Z_G * \lambda$
Delete()	$Z_C(1+(1-\alpha_C)*\lambda)$	$Z_T(1+(1-\alpha_T)*\lambda)$	$Z_G(1+(1-\alpha_G)*\lambda)$

Table 5: Worst case retrieval cost for Network Operations

The number of data pages retrieved is likely to be the lowest for CCAM in all cases, as it is likely to have the highest value for α. This situation is represented by $Z_C = Z_T = Z_G$. The total number of pages (i.e., index and data pages) retrieved by various methods shows more interesting trends. For simplicity, we will ignore the buffering effects in the following discussion.

To compare the constants Z_C and Z_T, we observe the following. Z_C is equal to 1 + the height of the secondary

index search tree in CCAM. Z_T represents 1 + the height of the primary index search tree in DFS-AM, assuming a primary index on the ordered node-id. In general, Z_C and Z_T are related by

$$Z_T = 1 + \log_{bfr_{index}} \frac{n}{bfr_{data}}$$

$$Z_C = 1 + \log_{bfr_{index}} n = Z_T + \log_{bfr_{index}} bfr_{data} \leq Z_T + 1 \tag{4}$$

$$ratio = \frac{Z_C - Z_T}{Z_C} = \frac{\log_{bfr_{index}} bfr_{data}}{1 + \log_{bfr_{index}} n} \rightsquigarrow 0 \; as \; n \rightsquigarrow \infty \tag{5}$$

where bfr_{index} denotes the average blocking factor for an index page in the B^+ tree, and bfr_{data} represents the average blocking factor for a data page. Equations 4 and 5 show that Z_T and Z_C differ at most by 1, and on average by $\log_{bfr_{index}} bfr_{data}$, which becomes a smaller and smaller fraction of Z_C as n increases. Thus the relative I/O cost of various operations in CCAM and DFS-AM will be dominated by the achievable CRR, as n increases. Usually, $\alpha_C \geq \alpha_T$, and thus CCAM is likely to have lower I/O costs than DFS-AM, for most operations, and for large networks.

In comparing the Grid File and CCAM, we observe the following. In general, Z_G is equal to 2. For large values of n, $Z_C > 2$. However, $\alpha_C \geq \alpha_G$ for most networks. For networks where proximity and connectivity are not correlated, CCAM is faster than the Grid File as $\alpha_C \gg \alpha_G$. For other networks, the Grid File may be faster. In those networks, CCAM can use Grid Files instead of the B^+ tree as the secondary index, to narrow and possibly close the performance gap.

The relative cost of the Delete() operation for alternative access methods shows the same patterns as the relative cost of the Get-successors() operation. The relative cost of the Insert() operation is predicted to be identical for all access methods, since the model cannot capture the clustering efficiencies for the neighbors of a new node being inserted. Even though connectivity-based methods (e.g. CCAM and DFS-AM) may not cluster these neighbors of a new node, spatial methods (e.g. the Grid File) are likely to cluster them well in networks where connectivity and proximity are correlated.

5 Experiment Design

Access methods for networks including CCAM and the update policies for CCAM, are evaluated by a series of experiments. In this section, we first describe the layout of our experiments and then illustrate the candidate access methods. Due to space constraints, we have only presented a subset of the experiments. A full description of these experiments and the results can be found in [30].

5.1 Experimental Layout

The design of our experiments is shown in Figure 6. We compare the proposed connectivity-based clustering scheme CCAM-S and CCAM-D with the other schemes, namely, the Grid file [31], the Cell Tree [17], Z-ordering [32], and DFS-AM [5, 28]. CCAM-S denotes the static create operation of CCAM. CCAM-D is an incremental create() operation which was implemented using the second-order reorganization policy. The Cell-tree represents the family containing R^+-tree [36]. The Grid-file and the Cell-tree partition the space to capture the isotropic nature of spatial proximity, which is an important property of spatial networks. We consider two versions of the Grid-file and Cell-tree, including connectivity-based and balance-based split policies, as described in Section 5.2. In Sections 6.1, 6.2 and 6.4, we use the connectivity-based split policies for Grid-file and Cell-tree.

Z-ordering [32] represents a spatial-based linear transformation of two-dimensional data. Topological ordering based methods (DFS-AM) are chosen for comparison, since they are the commonly used methods in the areas of path computations, transitive closures and recursive queries.

The effectiveness of the access methods will be evaluated based on the CRR (WCRR) values and I/O cost of route evaluation queries. We also evaluate the I/O cost for network operations and compare the experimental results with those for other methods.

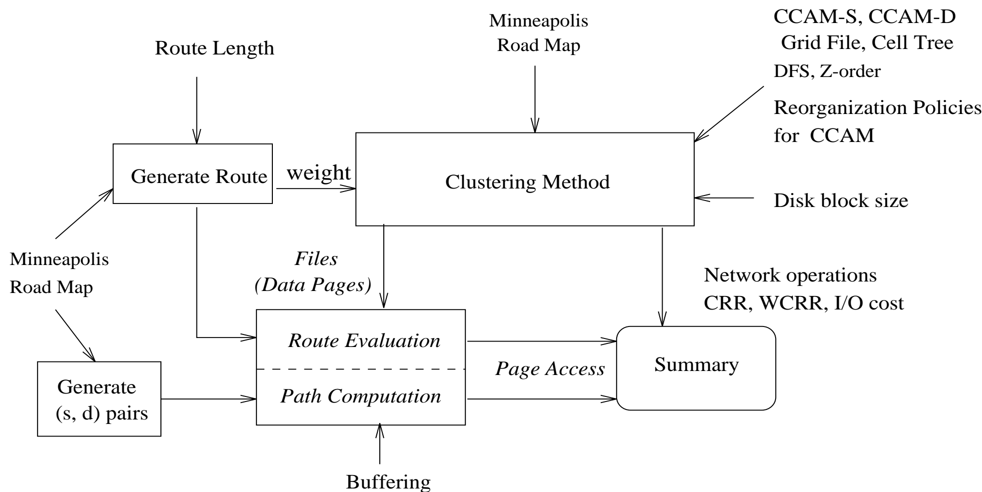


Figure 6: Experimental Layout

The experiments are conducted on many graphs. We present the results on a representative graph, which is a spatial network with 1079 nodes and 3057 edges that represents the major road intersections and highway segments for a 20-square-mile section of the Minneapolis area. This map is provided by the Minnesota Dept. of Transportation (MnDot). The data about each segment includes the x and y position of the two nodes, the average speed for the segment, average occupancy, and road type. The map is shown in Figure 7. The most

dense central region is at Minneapolis downtown. In this region, the roads run orthogonally or parallel to the river rather than north-south or east-west. It is interesting to note that even the outlying areas show a grid-like pattern of roads, except where lakes interrupt in the lower left corner, and where the Mississippi river flows (from north to southeast in the upper right quadrant of the map).

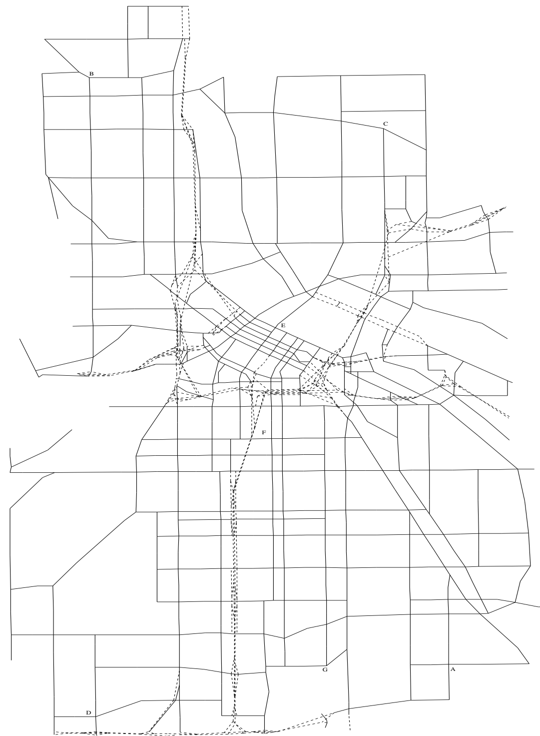


Figure 7: Minneapolis Road Map (Major roads)

We use a common record type for all the access methods. Each record contains a node and its neighbor-list, i.e., successor-list and predecessor-list. A node contains its (x, y) coordinates, and a neighbor-list contains a set of triples (x, y, attributes). Each triple represents the (x, y) coordinates of a neighbor of the node and the attributes of the edge connecting the node and the neighbor.

We conduct performance comparisons of I/O cost for network operations, I/O cost for route evaluation queries and path computation queries to evaluate the efficiency of various access methods. In addition, we also conduct experiments on the effect of split policies to demonstrate that spatial access methods can use connectivity-based split policies to increase the WCRR and thus increase performance. Table 6 summarizes the parameters explored in the experiments.

5.2 Candidate Access Methods other than CCAM

In this section we describe the candidate access methods used in the experiments.

Parameters	Values
Route Length	20, 40, 60 edges
Buffering	1, 8, 16
Disk Block Size	1/2, 1, 2, 4 K bytes

Table 6: Parameters

5.2.1 Grid File

The grid file [31] partitions the data space according to an orthogonal grid. The grid on a k-dimensional data space is defined by k one-dimensional arrays called scales. An element of a scale represents a k-1 dimensional hyperplane that partitions the space into two halves. There is 1-to-1 correspondence between the grid defined by the scales and the elements of a k-dimensional array called the grid directory. An element of this array holds a pointer to a disk block known as a data-page. This data-page contains the data points located in the corresponding grid cell. Low data-page utilization is avoided by allowing several grid cells to share a data page. The region of space occupied by the points stored in a page is called a data-page region. Data-page regions are rectangular boxes in k-dimensions. These regions are pairwise disjoint and their union spans the complete data space.

A common implementation of split policy in the Grid File evaluates two potential split points, one in the x-dimension and one in the y-dimension. Both of these points often evenly divide the records in a page, and either one may be chosen. We refer to the above approach as a balance-based split policy for Grid file. A connectivity-based policy for the Grid-file uses the connectivity information and chooses the split-dimension which has a higher WCRR.

5.2.2 Cell Tree

The cell tree [16, 17] is a height-balanced tree. Each cell tree node corresponds, not necessarily to a rectangular box, but to a convex polyhedron. The cell tree restricts the polyhedra to be partitions of a BSP (binary space partitioning), to avoid overlaps among sibling polyhedra. Each cell-tree node corresponds to one disk page, and the leaf nodes contain all the information required to answer a given search query.

The splitting of a cell tree node is based on the plane sweep paradigm, which conducts plane sweeps across the node along l different directions to find a suitable splitting hyperplane. A common split policy is to select a hyperplane that intersects a minimum number of cells and balances the two resulting subnodes. We refer to the above approach as the balance-based split policy for the Cell-tree.

We propose a new split policy, i.e., the connectivity-based split policy, to take advantage of connectivity

information. The hyperplane of choice should try to maximize the WCRR in the cell node. The splitting of a cell node N into N_1 and N_2 may now be accomplished by conducting plane sweeps across N along different directions to :

1. Find a hyperplane such that both subnodes can be stored on one disk page, and minimum page utilization constraint is satisfied.
2. Maximize the number of unsplit edges (or WCRR) with respect to the partitioning of N into N_1 and N_2.
3. In the case of a tie, choose the one that balances the resulting subnodes, i.e., $|sizeof(N_1) - sizeof(N_2)|$ is the smallest.

A simple implementation may use slope angles of the sweeping lines to be $\frac{i*180}{l}$ for $i = 1..l$, as suggested in [17]. We have tried various values for l and are currently using $l = 5$ plus the vertical sweep lines. A larger set of sweeping lines requires more computation time, but gains only slight improvements in performance.

5.2.3 Linear Clustering by Z-order

The Z-order [32] utilizes spatial information while imposing a total order on the points. The Z-order of a coordinate x,y is computed by interleaving the bits in the binary representation of the two values. Alternatively, Hilbert [12, 22, 24] ordering may be used. A conventional one-dimensional primary index (e.g. B^+-tree) can be used to facilitate searches.

5.2.4 Linear Clustering by DFS-order

DFS-AM arranges the nodes by a depth-first traversal from a random start node. This method extends the topological-ordering based method [28] to general graphs. A conventional one-dimensional primary index (e.g. B^+-tree) can be used to facilitate searches. DFS-AM is not the only method to linearly cluster data based on connectivity. We have also tried the Breadth First Search (BFS-AM) solution, which is implemented similarly but uses a breadth-first search. However, our results indicate that BFS-AM does not perform as well as DFS-AM. We therefore only report results from the DFS implementation.

6 Experimental Observations and Results

In this section, we present the results of our experiments, along with the effectiveness of the access methods and of the update policies that are based on measuring the CRR (WCRR) values and I/O costs. To simplify the comparison, the I/O cost represents the number of data pages accessed. This represents the relative performance

of the various methods for very large databases. For smaller databases, the I/O cost associated with the indices should be measured. In Sections 6.1, 6.4 and 6.5, we use a uniform weight (i.e., all weight on edges = 1) to simplify the interpretation of the results. In Sections 6.2 and 6.3, we use non-uniform weights on the edges (derived from a given set of routes). We examine the WCRR measure in the set of experiments that deals with route evaluation queries. Due to space constraints, we have only presented results from a subset of the experiments. A full description of these experiments and the results can be found in [30].

6.1 Evaluation of I/O Cost for Network Operations

We evaluate the I/O cost of alternative access methods for four network operations, namely, Get-A-successor(), Get-successors(), Insert(), and Delete(). The experiments use the Minneapolis road map with disk block size = 2 K. The cost for the Get-successors() operation is measured via performing the Get-successors() operation on a randomly chosen 50% of the total number of nodes. The cost for Get-A-successor() is computed similarly. Deletions are conducted on a randomly chosen 10% of the nodes. Insert() operations are conducted by inserting 10% of the nodes into a file created from the remaining 90% of the nodes in the Minneapolis road map. Page underflows and overflows in the Delete() and Insert() operations are ignored to filter out the effect of reorganization policies, which are studied separately. Table 7 shows the average number of data page accesses for each operation under various methods. The CRR value for each method is also listed in the table. The predicted cost for the Get-successors() and Get-A-successor() is computed using $(1-\alpha) * |A|$ and $1-\alpha$ respectively, as described in Section 4.2. The predicted cost for the delete operation is computed via the formula $2 * (1 + (1-\alpha) * \lambda)$.

Operation Method	Get-successors() Actual	Get-successors() Predicted	Get-A-successor() Actual	Get-A-successor() Predicted	Delete() Actual	Delete() Predicted	Insert() Actual	$\alpha =$ CRR
CCAM-S	0.418	0.413	0.147	0.145	2.935	2.933	4.187	0.8541
CCAM-D	0.454	0.494	0.167	0.174	3.109	3.118	4.504	0.8253
Cell Tree	0.606	0.700	0.248	0.247	3.499	3.583	3.701	0.7526
Grid File	0.642	0.736	0.267	0.260	3.588	3.664	3.401	0.7399
DFS-AM	0.770	0.893	0.284	0.315	3.966	4.018	4.149	0.6846
Zorder	0.928	1.077	0.371	0.380	4.107	4.433	3.495	0.6198
$\mid A \mid = 2.833 \quad \lambda = 3.20 \quad \gamma = 24.35$								

Table 7: The I/O costs of Network Operations for various access methods

As shown in Table 7, the number of data page accesses during Get-A-successor(), Get-successors() and Delete() operations with CCAM-S is the lowest among all the methods. This is to be expected, since CCAM-S has the highest CRR. The number of data page accesses in the Delete() operation is more than twice the number in the Get-successors() operation, because Get-successors() only retrieves the successor neighbors of a given node, while the Delete() operation updates ($Read\&Write$) both the successor and predecessor neighbors of a given node. Notice that in CCAM-S, the number of data page accesses in the Insert() operation is higher than the

number in the Delete() operation. This is because Delete() operations access the neighbors of a given node in a file, and those neighbors are likely to be put into the same disk page by CCAM-S to try to maximize the CRR. However, for the Insert() operation, there might exist few or no connections between the neighbors of the node to be inserted, and those neighbors might have been in different disk pages before the insertion. CCAM-S, CCAM-D and DFS-AM have higher I/O cost than proximity-based access methods, Cell-tree, Grid-file and Z-ordering in the Insert() operation. The spatial proximity of the neighbors of the new node to be inserted helps the Cell-tree, Grid file and Z-ordering reduce the I/O cost of the Insert() operation. The Z-ordered index of CCAM does not help because it is a secondary index.

6.2 Aggregate Query : Route Evaluation

To evaluate the performance of alternative access methods on aggregate queries over networks, we work with route evaluation queries. We generate routes by performing random walks on the network. The weights on the edges of the network are derived by counting the number of times that an edge is accessed by a set of routes. A route of length L has L nodes and $L-1$ edges. We generate three sets of routes with lengths equal to 20, 40, and 60 edges respectively. Each set contains 100 routes. Six alternate methods are used to store the road maps based on the weight created, and 300 route evaluation queries are performed to compare the number of data page accesses. The route evaluation queries are processed by issuing a Find() operation followed by a sequence of Get-A-successor() operations.

6.2.1 The Effect of Route Length

In this subsection, we report the results obtained with minimum buffering, i.e., a buffer with one data page. To examine the effect of route length on the I/O cost, we plot the detailed result for different route lengths. Figures 8 (a) and (b) show the results of the experiments conducted on block size 512 and 4096 bytes, respectively. The number of data page accesses for route evaluation queries decreases with the increase of block size, for all methods. The number of data page accesses increases linearly with route length L, as predicted by the cost models. CCAM-S and CCAM-D outperform all the other methods. The Cell-tree ranks next-best for large block sizes, while DFS-AM ranks next-best for smaller block sizes.

6.2.2 Does the WCRR Predict the Cost of Route Evaluation?

Figures 9 (a) and (b) show the average number of data pages accessed per route (averaged over 300 routes) and the WCRR respectively, for various methods, as the block sizes change. The average number of nodes accessed per route evaluation query is equal to 40. We observe that a higher WCRR implies a lower number of data page

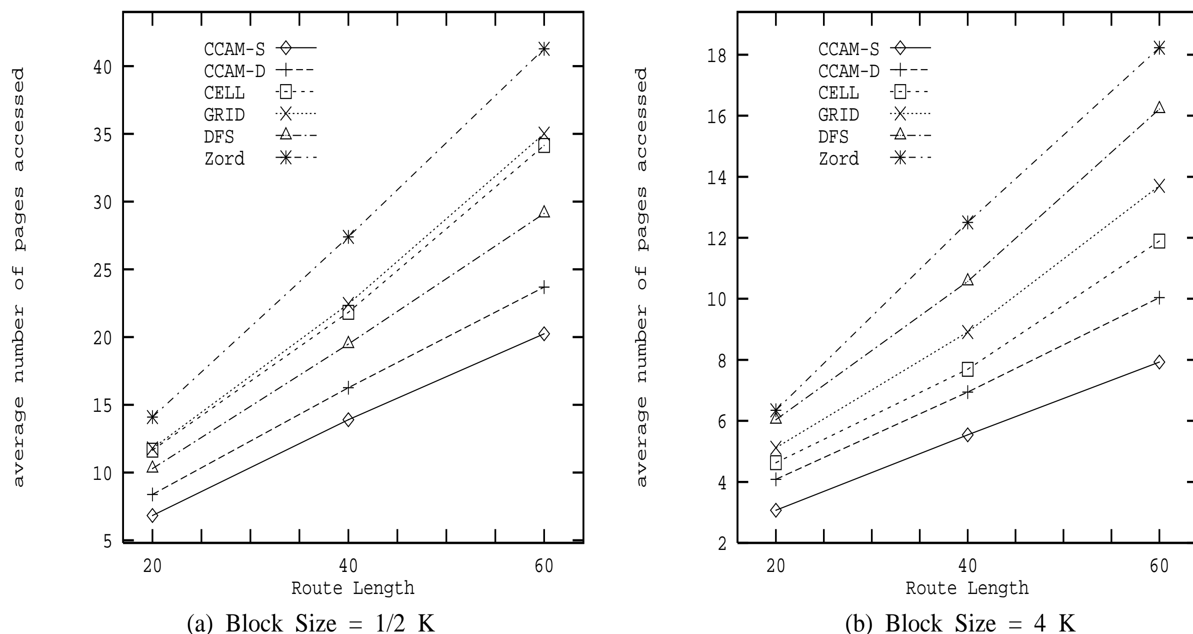


Figure 8: Effect of route length (number of buffers = 1)

accesses for route evaluation queries, as predicted by the cost models. CCAM-S and CCAM-D outperform the others consistently for all four block sizes. The Grid file and Cell-tree perform worse than DFS-AM for smaller block sizes, but they perform better than DFS-AM for larger block sizes.

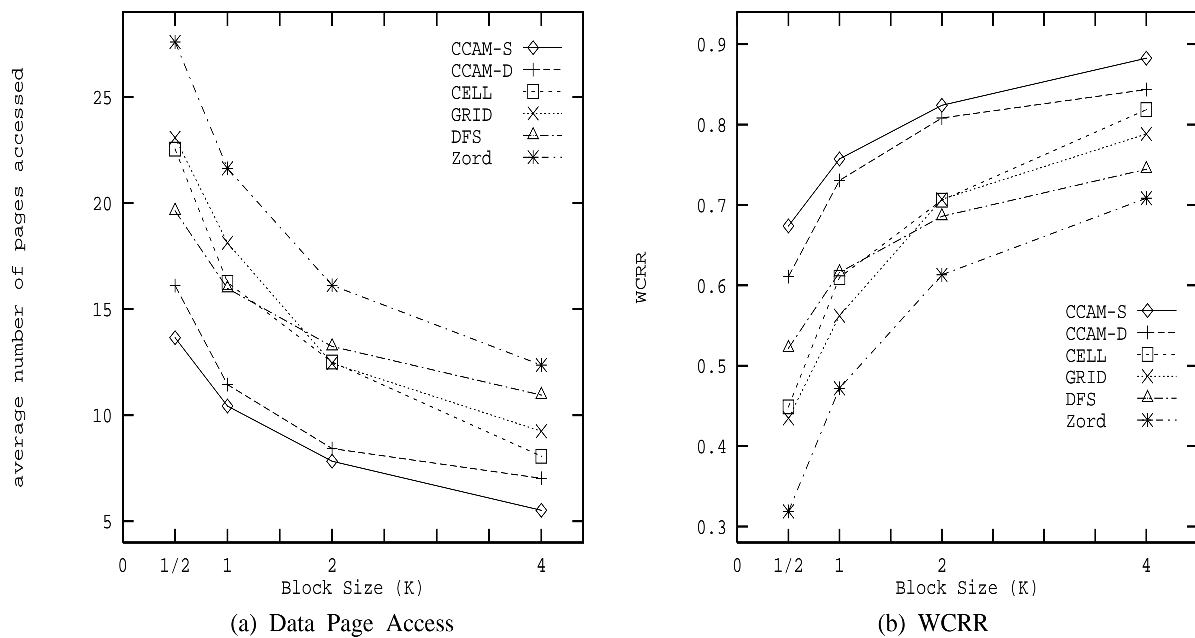
6.2.3 The Effect of Buffering

In this section, we evaluate the effect of buffering on the performance of the access methods. The variable parameters are the number of buffers available. The experiments are performed using route evaluations as the benchmark queries. We report the average I/O over 300 paths for the real Minneapolis road map.

Figure 10 shows the effect of buffering on the performance of route evaluation, on the Minneapolis road map, for various access methods with disk block size 1 K. We observe an improvement in performance as the number of buffers increases. The performance ranking for each access method remains the same for different numbers of buffers.

6.3 Do the Proposed Split Policies Help Spatial Access Methods?

The experiment on the effect of the different split policies is conducted using the route evaluation queries and the Minneapolis road map. Figures 11 (a) and (b) show the average number of data pages accessed per route (averaged over 300 routes) and the WCRR respectively, for the Cell tree and Grid file. CELL-C and GRID-C



(a) Data Page Access

(b) WCRR

Figure 9: Average I/O costs per route and WCRR (number of buffers = 1)

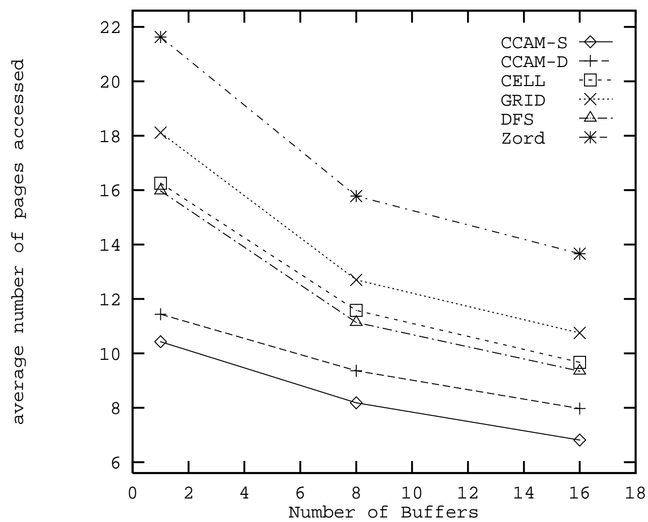
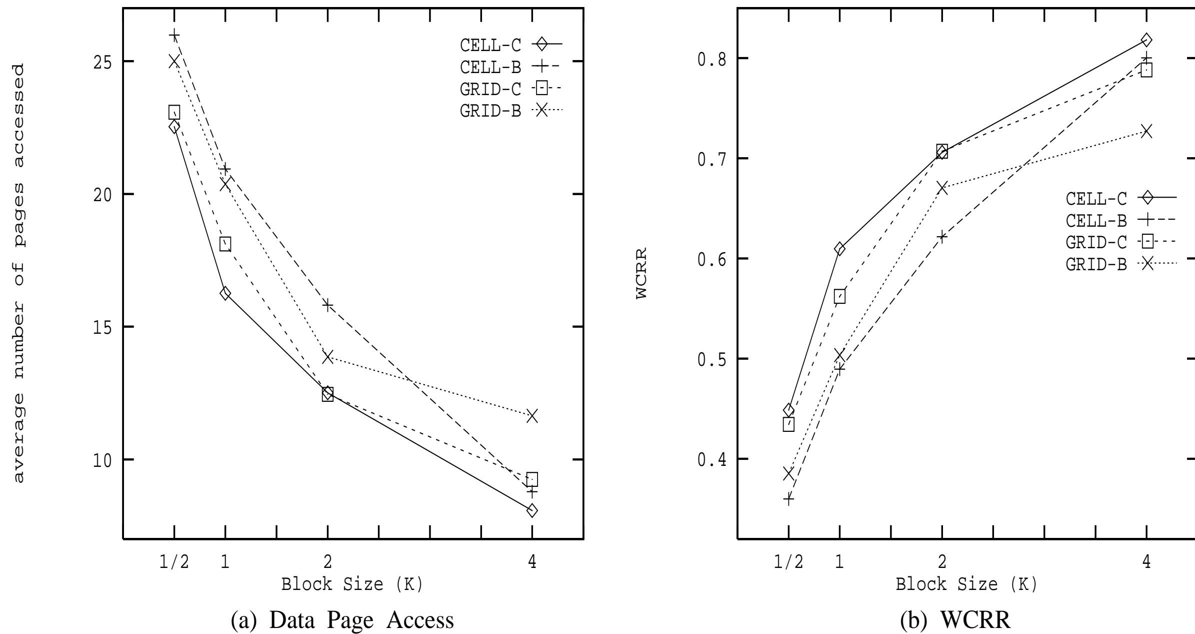


Figure 10: Effect of buffering on data page accesses (block size = 1 K)

denote the Cell-tree and Grid-file with the connectivity-based split policy. CELL-B and GRID-B denote the Cell-tree and Grid-file using the balance-based (non-connectivity) split policy. The experiment is conducted with disk block size 1 K and 1 buffer.

The Cell-tree with the connectivity-based split policy (CELL-C) has a higher WCRR and a lower number of data page accesses than those of the Cell-tree with the balance-based split policy (CELL-B). The Grid-file shows a similar trend. The Grid-file with the connectivity-based split policy (GRID-C) has a higher WCRR and a lower number of data page accesses than those of the Grid-file with the balance-based split policy (GRID-B). Thus, spatial access methods may utilize connectivity information to better serve spatial networks and network computations.



(a) Data Page Access

(b) WCRR

Figure 11: Effect of Split Policy

6.4 Evaluation of CCAM for Path Computation

The evaluation of I/O cost for path computation is conducted using the path computation algorithm, the $A^\star$ algorithm, with the Euclidean distance heuristic. $A^\star$ represents the single-pair path-planning algorithms which use heuristic lookahead to focus the search [9, 38]. Three query sets are chosen to represent path queries of three different path classes, namely small, medium and large. The small, medium and large classes include 25 randomly chosen (source, destination) pairs with the route length, i.e., the number of edges on the shortest path, equal to 8 ± 2, 23 ± 2 and 34 ± 2, respectively. Our metric of comparison was the number of data pages accessed. We do

not count the index pages accessed as part of the cost. Experiments are conducted using disk block size 2 K. A simple implementation of $A^{\star}$ is used, with no special data structure to manage information inside memory. The integration of CCAM with more sophisticated implementations of path computation algorithms is desirable, and we will explore this in future work.

The effect of route length on I/O cost for the six access methods is shown in Figures 12 (a) and (b) for 1 and 8 buffers, respectively. Figures 12 (a) and (b) show the mean number of pages accessed by the $A^{\star}$ algorithm for three route length classes: small, medium and large. We can see an increase in the number of pages accessed as the length of the path increases. CCAM-S and CCAM-D outperform the others consistently for all three route length classes. In general, the I/O cost ranking for all six access methods is the same for all three route length classes, but the I/O gap between the different access methods increases as the route length increases. The CRR values for the various access methods are listed in Table 7. As we expected, access methods with a higher CRR have a lower I/O cost for path computation. The performance ranking for each access method remains the same for 1 and 8 buffers.

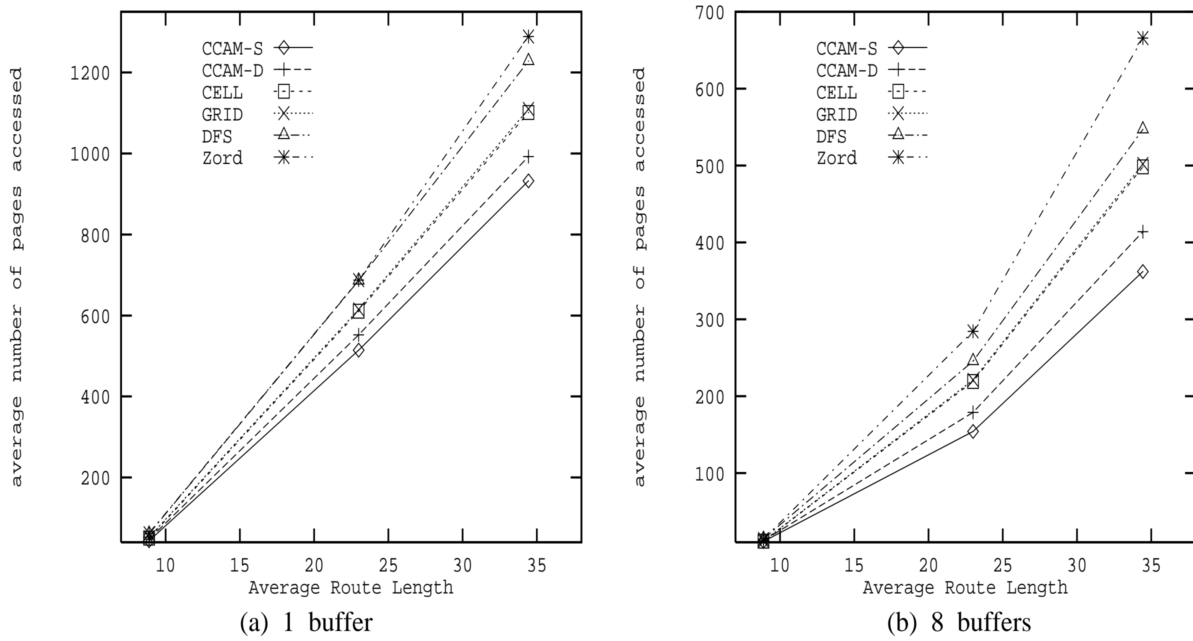


Figure 12: Performance Comparison for Path Computation (block size = 2 K)

6.5 Evaluation of Reorganization Policies

Update Operations : CRR vs Update I/O Cost

Figure 13 shows the I/O cost and CRR results during the insertion of 20% of the nodes of the Minneapolis

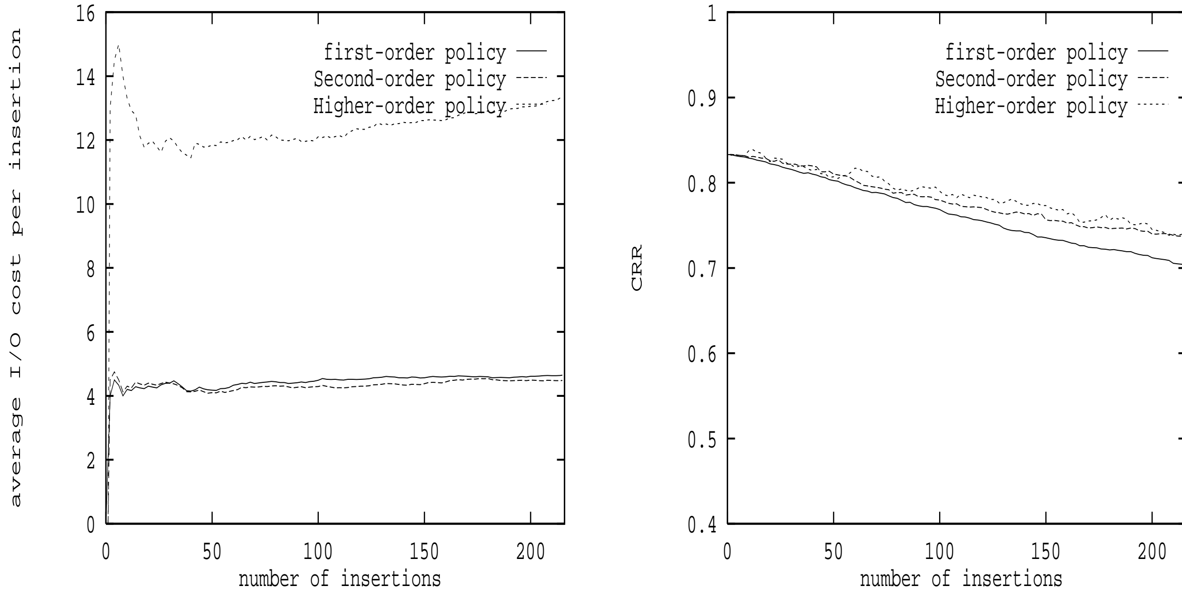


Figure 13: Effect of the reorganization policies (insert)

road map, using a random insertion order and block size 1 K. It shows that the higher order policy has a much higher I/O cost than the first-order and second-order policies. The average I/O costs for the first-order and second-order policies are very close, as expected. Notice that the average I/O cost for the higher-order policy increases slightly as the number of insertions increases. This is to be expected, since the CRR value decreases as the number of insertions increases, resulting in an increase in the number of data pages accessed for neighboring pages. The CRR might increase during some insertions, but it decreases in the long run, because the average connectivity increases and the average blocking factor is reduced as more nodes are inserted into the file. In most cases, The first-order policy has the lowest CRR, and the higher-order policy has a slightly higher CRR than the second-order policy. The second-order policy is a possible trade-off choice, since its I/O cost is almost the same as that of the first-order policy, and its CRR is competitive with the higher-order policy. We note that the choice of reorganization policy depends on the relative frequencies of update and retrieval operations, as discussed in Section 3.4.

7 Conclusions and Future Work

Network computations, including route evaluation and related aggregate queries on networks, are used in many important applications of databases such as IVHS. It is important to design storage and access methods to support these applications. We have identified a measure, namely the WCRR, that is able to accurately predict

the performance of an access method for network computations. An algebraic cost model for network operations has been derived. The analysis shows that the efficiency of the Delete(), Get-A-successor() and Get-successors() operations depends primarily on the CRR.

We have presented a connectivity-clustered access method (CCAM) for general networks to support network operations and aggregate queries over networks. We also identify and evaluate alternate reorganization policies for CCAM to maintain a high WCRR, without incurring high reorganization costs, during insertion and deletion of nodes and edges.

We have evaluated alternative access methods for network computations. Applications that require the most efficient processing of network computations should use CCAM, which achieves the highest WCRR and provides the best performance. However, access methods based on spatial proximity can also be improved if the WCRR is used to develop new split policies.

Future work includes developing a formal analysis for achievable CRR under different access methods. A more efficient index access structure should be designed that will efficiently support incremental reorganization during update operations. Further, the CPU cost for reorganization may be taken into account. Reorganization policies that do not incur high CPU costs are currently being investigated. Finally, we would like to evaluate CCAM for other aggregate queries on networks, including tour evaluation and location-allocation evaluation, as well as on mixed workloads such as the sequoia benchmark [40].

8 Acknowledgment

This research was supported by the Federal Highway Authority (FHWA), Minnesota Dept. of Transportation and the Center for Transportation Studies at the University of Minnesota. We would like to thank Dr. H.V. Jagadish (AT&T Bell Labs) and Prof. K. Hua (University of Florida) for helping with the survey and focus of this research. We would also like to thank Prof. C.K. Cheng (University of California, San Diego) and Dr. L.T. Liu (AT&T Bell Labs) for helping with the ratio-cut program. Finally, the interaction with the Network Engine group at Environmental Systems Research Institute helped in assessment of many ideas for suitability to GIS software systems such as ARC/INFO.

References

[1] "Intelligent Vehicle Highway Systems Projects". Department of Transportation, Minnesota Document, March 1994.

[2] D. Abel and D. Mark. "A Comparative Analysis of Some Two-Dimensional Orderings". *Intl Journal of GIS*, 4(1):21–31, 1990.

[3] R. Agrawal and H.V. Jagadish. "Algorithms for Searching Massive Graphs". *IEEE Trans. on Knowledge and Data Engineering*, 6(2), April 1994.

[4] R. Agrawal and Jerry Kiernan. "An Access Structure for Generalized Transitive Closure Queries". In *Proc. of the Ninth Intl Conference on Data Engineering*, pages 429–438. IEEE, April 1993.

[5] J. Banerjee, S. Kim W. Kim, and J. Garza. "Clustering a DAG for CAD Databases". *IEEE Trans. on Software Engineering*, 14(11):1684–1699, Nov. 1988.

[6] E.R. Barnes. "An Algorithm for Partitioning the Nodes of a Graph". *SIAM Journal Alg. Disc. Meth.*, 3(4):541–550, December 1982.

[7] C.K. Cheng and Y.C. Wei. "An Improved Two-Way Partitioning Algorithm with Stable Performance". *IEEE Trans. on Computer-Aided Design*, 10(12):1502–1511, December 1991.

[8] W. C. Collier and R. J. Weiland. "Smart Cars, Smart Highways". *IEEE Spectrum*, pages 27–33, April 1994.

[9] D. Galperin. "On the optimality of A*". *Artificial Intelligence*, 8(1):69–76, 1977.

[10] S. Dar and H.V. Jagadish. "A Spanning Tree Transitive Closure Algorithm". In *Proc. of Intl Conference on Data Engineering*. IEEE, 1992.

[11] C. Faloutsos and Y. Rong. "DOT: A Spatial Access Method Using Fractals". In *Proc. of the 7th Intl Conference on Data Engineering*. IEEE, 1991.

[12] C. Faloutsos and S. Roseman. "Fractals for Secondary Key Retrival". In *Proc. Symp. on Principles of Database Systems*. SIGMOD-SIGACT PODS, 1989.

[13] C.M. Fiduccia and R.M. Mattheyses. "A Linear Time Heuristic for Improving Network Partitions". In *Proc. of 19th Design Automation Conference*, pages 175–181, 1982.

[14] M.R. Garey and D.S. Johnson. *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*. W.H. Freeman and Company, San Francisco, 1979.

[15] M.F. Goodchild. "Towards an Enumeration and Classification of GIS Functions". In *Proc. of Intl Geographic Info. Systems Symp.*, 1987.

[16] O. Gunther. "The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases". In *Proc. 5th Intl Conference on Data Engineering*, Feb. 1989.

[17] O. Gunther and J. Bilmes. "Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation". *IEEE Trans. on Knowledge and Data Engineering*, 3(3), September 1991.

[18] A. Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". In *Proc. of SIGMOD Intl Conference on Management of Data*, pages 47–57. ACM, 1984.

[19] P. Haggett and R. J. Chorley. *"Network Analysis in Geography"*. St. Martin's Press, New York, 1969.

[20] K. Hua, J. Su, and C. Hua. "An Efficient Strategy for Traversal Recursive Query Evaluation". In *Proc. of the Ninth Intl Conference on Data Engineering*. IEEE, April 1993.

[21] Y. Ioannidis, R. Ramakrishnan, and L. Winger. "Transitive Closure Algorithms Based on Graph Traversal". *ACM Trans. on Database Systems*, 18(3), September 1993.

[22] H.V. Jagadish. "Linear Clustering of Objects with Multiple Attributes". In *Proc. of Intl Conference on Management of Data*, pages 332–342. ACM SIGMOD, 1990.

[23] B. Jiang. "I/O Efficiency of Shortest Path Algorithms: An Analysis". In *Proc. of the Intl Conference on Data Engineering*. IEEE, 1992.

[24] I. Kamel and C. Faloutsos. "Hilbert R-tree: An Improved R-tree using Fractals". In *Proc. of Intl Conference on Very Large Data Bases*, 1994.

[25] B.W. Kernighan and S. Lin. "An Efficient Heuristic Procedure for Partitioning Graphs". *Bell Syst. Tech. J.*, 49(2):291–307, February 1970.

[26] R. Kung, E. Hanson, and et. al. "Heuristic Search in Data Base Systems". In *Proc. Expert Database Systems*. Benjamin Cummings Publications, 1986.

[27] Y. Kusumi, S. Nishio, and T. Hasegawa. "File Access Level Optimization Using Page Access Graph on Recursive Query Evaluation". In *Proc. Conference on Extending Database Technology*. EDBT, 1988.

[28] P.A. Larson and V. Deshpande. "A File Structure Supporting Traversal Recursion". In *Proc. of the SIGMOD Conference*, pages 243–252. ACM, 1989.

[29] R. Laurini and D. Thompson. *"Fundamentals of Spatial Information Systems"*, chapter 5 and 2.5.4. Number 37 in The A.P.I.C. Series. Academic Press, 1992.

[30] D.R. Liu and S. Shekhar. "CCAM: Connectivity-Clustered Access Method for Networks and Network Computations". Technical Report TR 93-78, Computer Science Dept. University of Minnesota, 1993.

[31] J. Nievergelt, H. Hinteberger, and K.D. Sevcik. "The Grid File: An Adaptable, Symmetric Multi-Key File Structure". *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

[32] A. Orenstein and T. Merrett. "A Class of Data Structures for Associative Searching". In *Proc. Symp. on Principles of Database Systems*, pages 181–190. SIGMOD-SIGACT PODS, 1984.

[33] J.T. Robinson. "The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic indexes". In *Proc. of SIGMOD Intl Conference on Management of Data*, pages 10–18. ACM, 1981.

[34] H. Samet. *"The Design and Analysis of Spatial Data Structures"*. Addison Wesley, 1990.

[35] P. Scheuermann and M. Ouskel. "Multidimensional B-trees for Associative Searching in Database Systems". *Information Systems*, 7(2), 1982.

[36] T. Sellis, N. Roussopoulos, and C. Faloutsos. "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects". In *Proc.13th Intl Conference on Very Large Data Bases*, pages 507–518, 1987.

[37] S. Shekhar, A. Kohli, and M. Coyle. "Can Proximity-Based Access Methods Efficiently Support Network Computations?". Technical Report TR-93-57, Computer Science Dept. University of Minnesota, 1993.

[38] S. Shekhar, A. Kohli, and M. Coyle. "Path Computation Algorithms for Advanced Traveler Information System". In *Proc. of the Ninth Intl Conference on Data Engineering*, pages 31–39. IEEE, April 1993.

[39] S. Shekhar and D. R. Liu. "Genesis and Advanced Traveler Information Systems (ATIS) : Killer Applications for Mobile Computing". In *NSF MOBIDATA Workshop on Mobile and Wireless Information Systems*, 1994.

[40] M. Stonebraker, J. Frew, K. Gardels, and Jeff Meredith. "The Sequoia 2000 Benchmark". In *Proc. of Intl Conference on Management of Data*. ACM, 1993.

[41] M. M. Tsangaris and Jeffrey.F. Naughton. "A Stochastic Approach for Clustering in Object Bases". In *Proc. of SIGMOD Conference on Management of Data*, pages 12–21. ACM, 1991.

[42] P. Valduriez and H. Boral. "Evaluation of Recursive Queries Using Join Indices". In *Proc. of Intl Conference on Expert Database Systems*, 1986.

[43] A. P. Vonderohe and et. al. "Adaption of GIS for Transportation". NCHRP Report 359, Transportation Research Board, National Research Council, 1993.

[44] J. L. Wiener and J. F. Naughton. "Bulk Loading into an OODB: A Performance Study". In *Proc. of Intl Conference on Very Large Data Bases*, 1994.

[45] C.W. Yeh, C.K. Cheng, and T.T. Y. Lin. "A General Purpose Multiple Way Partitioning Algorithm". In *Proc. of 28th ACM/IEEE Design Automation Conference*, pages 421–426, 1991.

A Choosing a Heuristic for Graph Partitioning

Cheng and Wei [7] have shown that the quality of two-way partitions can be improved by incorporating the balancing of partition sizes in the cost metric, rather than by imposing constraints on the partition sizes. They define the ratio cost metric for a two-way partition as $E_c/|A| * |B|$, where E_c is the sum of the weights of the edges cut, and $|A|$ and $|B|$ are the sizes of the two partitions.

The philosophy of the ratio cut is to identify the natural clusters in the graph. First, they remove the constraint on subset size. The algorithm dynamically establishes its own subsets, which are close to natural clusters in the graph. It consists of three major phases: 1) initialization, 2) iterative shifting, and 3) group swapping. Their implementation of the above technique is based on the bucket list data structure proposed by Fiduccia and Mattheyses [13]. Second, the size constraints on the resultant subsets are enforced by applying the Fiduccia-Mattheyses algorithm to fine-tune the final result.

We have been inspired by the objective function of the ratio cut partitioning, which successfully embodies both the min-cut and equipartition goals of partitioning. In our experiment, we adapt their two-way ratio-cut algorithm as the basis for our connectivity-based clustering method. In this paper, we abstract two-way ratio-cut graph partitioning as the following procedure.

2-way-partition:(V: set of nodes; E: set of edges; min-page-size) → <A1, A2>
where A1 and A2 are set of nodes,
sizeof(A1) > min-page-size, sizeof(A2) > min-page-size and sizeof(V) > 2*min-page-size.

B Cost Modeling Framework for Update Operations

In this section, we illustrate the simple algebraic I/O cost model for update operations. To simplify our cost modeling, we assume that the index pages are buffered in the main memory and that sufficient buffers are provided for the update operations. Thus, we will only focus on the number of data pages accessed in each update operation.

B.1 Cost model for the Insert() Operation

The cost for the Insert() operation mainly consists of retrieving those pages which contain nodes that are neighbors of the given new node to be inserted. This step is necessary to maintain successor-lists and predecessor-lists of neighbors that are consistent with those of the newly inserted node. The cost of fetching the neighboring nodes of a given new node is modeled as follows. Let f^I_{ave}, f^I_{min}, f^I_{max} respectively be the average, minimum and maximum number of distinct pages which contain nodes that are neighbors of a new node to be inserted, then

$$f^I_{min} = [\frac{\lambda}{\gamma}] \leq f^I_{ave} \leq [\lambda] = f^I_{max} \quad (6)$$

We use [x] to denote the function max(x, 1). The function [x] is used, since at least one data page access is required in any case. Equation 6 shows that in the best case, as many neighbors as possible are located in the same data page; thus f^I_{min} is equal to $[\lambda/\gamma]$. Notice that the neighboring nodes of the newly inserted node might be distributed across data pages without any connection. In the worst case, each of the neighbors may be located in a different page, so f^I_{max} is equal to $[\lambda]$. Obviously, f^I_{ave}, f^I_{min}, f^I_{max} also represent the average, minimum and maximum *Read* cost of the first-order and second-order policy. For the higher-order policy, additional retrieval is needed for fetching neighboring pages of P, where P is the page chosen in which to place the newly inserted node. Let h^I_{ave}, h^I_{min}, h^I_{max} respectively be the average, minimum

and maximum number of pages which are neighboring pages of P in the page access graph, then

$$h^I_{min} = [\frac{\gamma * \lambda * (1-\alpha)}{\gamma}] \leq h^I_{ave} \leq [\gamma * \lambda * (1-\alpha)] = h^I_{max} \tag{7}$$

Considering a node x in P, $\alpha * \lambda$ neighbors of x are in the same page as x (assuming $\gamma > \alpha * \lambda$), and the other $\lambda * (1-\alpha)$ neighbors are in different pages. Since the average blocking factor is γ, the total number of neighboring nodes located outside page P is equal to $\gamma * \lambda * (1-\alpha)$. It is clear that h^I_{min} is equal to $[\lambda * (1-\alpha)]$, and h^I_{max} is equal to $[\gamma * \lambda * (1-\alpha)]$. The average, minimum and maximum *Read* cost for the higher-order policy can be expressed by the sum of equation 6 and equation 7.

The total number of page-writes ($Write$ cost) is equal to the total number of page-reads unless there is overflow. In general, the $Write$ operation will require an additional page if overflow occurs. Suppose the probability of overflow is ρ, then the $Write$ cost is equal to the *Read* cost * $(1-\rho)$ + (*Read* cost + 1) * ρ. We can then derive the cost of $Write$, which is equal to *Read* cost + ρ.

B.2 Cost model for Delete() Operation

Like the Insert() operation, the Delete() operation also needs to maintain the successor-lists and predecessor-lists of neighboring nodes. The retrieval cost of the update operation is modeled as follows. Let f^D_{ave}, f^D_{min}, f^D_{max} respectively be the average, minimum and maximum number of distinct pages that contain the neighboring nodes of a given node x, which exists in the data page, then

$$f^D_{min} = [1 + \frac{\lambda * (1-\alpha)}{\gamma}] \leq f^D_{ave} \leq [1 + \lambda * (1-\alpha)] = f^D_{max} \tag{8}$$

There are $\alpha * \lambda$ neighbors in the same page as x; thus one page is needed for them. For the other $(1-\alpha) * \lambda$ neighbors, they can be located at least in $(1-\alpha) * \lambda/\gamma$ pages and at most in $(1-\alpha) * \lambda$ pages. This explains the f^D_{min} and f^D_{max} values in equation 8, since one data page access is required to retrieve the node to be deleted, and $\alpha * \lambda$ neighbors are in the same page as the node to be deleted. Therefore, f^D_{min}, f^D_{ave} and f^D_{max} represent the *Read* cost for the first-order and second-order policies. For the higher-order policy, the additional cost required for retrieving the neighboring pages can be modeled using an analysis similar to the one used in the cost modeling of the Insert() operation, except that the maximum number of nodes in P is $(\gamma - 1)$, since one node is deleted from page P.

Let h^D_{ave}, h^D_{min}, h^D_{max} respectively be the average, minimum and maximum number of pages which are neighbors of P in the page access graph, and then

$$h^D_{min} = [\frac{(\gamma-1) * \lambda * (1-\alpha)}{\gamma}] \leq h^D_{ave} \leq [(\gamma-1) * \lambda * (1-\alpha)] = h^D_{max} \tag{9}$$

The *Read* cost of the higher-order policy is equal to the sum of equation 8 and equation 9. The number of page-writes ($Write$ cost) is equal to the number of page-reads, unless there is underflow. In general, the $Write$ cost will be one less page-write if underflow occurs. Suppose that the probability of underflow is μ; then the $Write$ cost is equal to the *Read* cost * $(1-\mu)$ + (*Read* cost - 1) * μ. We can derive the cost of $Write$, which is equal to *Read* cost - μ. Since it is very difficult to analytically compute the probability of underflow, we will assume that μ is zero in our comparison of predicted cost to actual cost.