Technical Report

Department of Computer Science
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 97-003

Materialization Trade-Offs in Hierarchical Shortest Path Algorithms: A Case Study in ATIS

by: Shashi Shekhar, Andrew Fetterer, and Brajesh Goyal

Materialization Trade-Offs in Hierarchical Shortest Path Algorithms: A Case Study in ATIS [1]

Shashi Shekhar, Andrew Fetterer, Brajesh Goyal

Computer Science Department
EE/CS 4-192, 200 Union St. SE., University of Minnesota, Minneapolis, MN. 55455

[*shekhar*|*fetterer*|*brajesh*]@cs.umn.edu
http://www.cs.umn.edu/research/shashi-group/
Voice: (612) 624-8307 Fax: (612) 625-0572

Abstract

Materialization and hierarchical routing algorithms are becoming important tools in querying databases for the shortest paths in time-critical applications like Intelligent Transportation Systems (ITS), due to the growing size of their spatial graph databases [23]. A hierarchical routing algorithm decomposes the original graph into a set of fragment graphs and a boundary graph which summarizes the fragment graphs. A fully materialized hierarchical routing algorithm pre-computes and stores the shortest-path view and the shortest-path-cost view for the graph fragments as well as for the boundary graph [13]. The storage cost of the fully materialized approach can be reduced by a virtual or a hybrid materialization approach, where few or none of the relevant views are pre-computed. This paper explores the effect of materializing individual views for the storage overhead and computation time of hierarchical routing algorithms. Our experiments with the Twin Cities metropolitan road-map show that materializing the shortest-path-cost view for the boundary graph provides the best savings in computation time, for a given amount of storage and a small number of fragments. Materializing the relevant part of the shortest-path-cost view for the fragment graphs provides the next best savings, followed by materializing the shortest-path view for the boundary graph. Virtual shortest-path-view on fragments can reduce storage costs by an order of magnitude or more for large graphs.

1 Introduction

Spatial graph databases [23] form the kernel of many important applications including transportation; 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 graphs which consist of a finite collection of 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 graph database for transportation applications may store road-intersection points and road segments connecting the intersections. Spatial graph databases for large cities (e.g. road-maps) contain millions of nodes and edges.

Shortest-path queries retrieve a sequence of nodes and edges forming an optimal (e.g. shortest distance or travel time) path between the given node-pair (source,destination). The shortest-path-cost query computes the aggregate cost (e.g. distance) of the shortest-path. Materialization and hierarchical algorithms are becoming important in processing shortest-path queries in time-critical applications such as Intelligent Transportation Systems [20, 25, 4]. Hierarchical algorithms decompose a large spatial graph into a boundary

[1]This work was supported by the NSF(grant number NSF/IRI-9631539) and the Federal Highway Authority(Minnesota Guidestar Program)

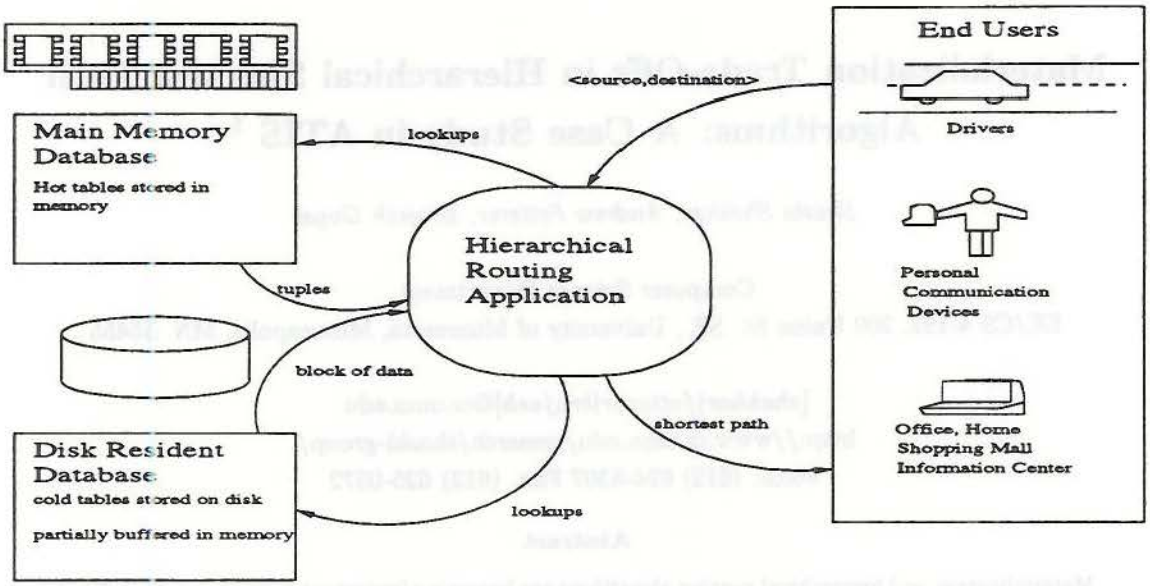


Figure 1: MMDB and DRDB Components of an ATIS Database

graph and a collection of fragment graphs, each of which is smaller than the original graph. The shortest-path query on the large graph is decomposed into a collection of shortest-path-cost queries and shortest-path queries on the smaller graphs.

1.1 ATIS Architecture

Advanced Traveler Information Systems (ATIS) assist travelers with planning, perception, analysis and decision-making to improve the convenience, safety and efficiency of travel [20, 22]. ATIS is one facet of the Intelligent Transportation System (ITS), which is currently being developed to improve the safety and efficiency of automobile travel. Path-planning is an essential component of ATIS, aiding travelers in choosing the optimal path to their destinations in terms of travel distance, travel time and other criteria. Estimates made by government agencies show that approximately 6 percent of all driving time in the U.S. is due to an incorrect choice of paths.

Due to the dynamic nature of the ITS database and the frequency and nature of queries, it is necessary to explore alternatives to traditional disk-resident databases (DRDB). Main-memory databases (MMDB) [8] store most of their content permanently in main memory, resulting in high speed access to that data. Data stored in memory is often *hot* if it is frequently accessed, while data kept on disk is often *cold* if it is accessed less frequently. Main memory databases can store both types of data. MMDBs differ from DRDBs with large caches, since the MMDB does not have to treat the data as if it were stored on disk (i.e. deal with index structures and the buffer manager). Storing the data permanently in memory eliminates the need for clustering since since random access is no more costly than sequential access. Since disk access time is not a concern for the hot data, performance is also dependent upon processing time.

Figure 1 shows the flow of information in a typical ATIS database consisting of a MMDB and a DRDB. Users access ATIS data through in-vehicle communications, personal communications devices, or over the Internet. The users query the system with the source and destination of their route. The hierarchical routing algorithm partitions its data (e.g. road maps, fragments, boundary graph, and various other views) between its main memory database and its disk resident database. The fragment graphs and boundary-graph for

metropolitan areas are kept in the MMDB if adequate main-memory is available. The MMDB also holds smaller views that are accessed frequently. The DRDB contains other views. A shortest path query on the base graph is decomposed into shortest-path-cost queries and shortest-path queries on fragments and the boundary graph. A query on fragments or the boundary graph can be answered immediately with lookups, if relevant materialized views are available either in MMDB or DRDB. Otherwise, those queries are answered with on-line computation.

Some of the issues in the performance tuning of the hierarchical routing system include (i) the choice of views to materialize and (ii) the partitioning of those views between MMDB and DRDB. Other issues relate to the decomposition of graphs into fragments, to the choice of the number of levels in the hierarchy, and to view maintenance strategies [13]. Other designs of ATIS databases are possible, however, we focus our discussion within the context of the design described in Figure 1.

1.2 Related Work and Our Contributions

The single-pair path-computation problem is a special case of single-source path computation and all-pair path computation. Traditional research in database query languages [19, 1] , transitive closure [2, 11, 10] and recursive query processing [17] has approached single-pair path-computation as a special case of more general problems. For example, partial-transitive-closure computation [12] and transitive-closures [11] have been used for single-pair path computations. Previous evaluation of the transitive-closure algorithms examined the iterative, logarithmic, Warren's, Depth-first search (DFS), hybrid, and spanning-tree-based algorithms [18, 3, 5].

Previous evaluation of single-pair path-computation algorithms in a database environment examined the iterative breadth-first and best-first search algorithms [16]. Another study [21] examined A* and other estimator-based algorithms, along with the effect of path lengths and edge costs on the relative performance of search algorithms. Query-language-based implementations were found to be inefficient [16, 21].

Hierarchical path-finding has been explored in the context of computer networks in [15] and in the context of planar graphs in [6, 7]. In ATIS, a hierarchical routing algorithm called HEPV [13] that guarantees optimal solutions has been investigated. That hierarchical routing algorithm divides the base graph into a boundary graph and a set of fragment graphs. It has materialized views for the shortest-path and shortest-path-cost for the fragments and the boundary graph. The storage of these views reaches gigabytes for a graph with 100,000 nodes and may become prohibitively large for ATIS maps with millions of nodes. Selective materialization can be used to manage storage costs. That work also explored performance issues such as the decomposition of graphs into fragments, the number of levels in the hierarchy, clustering of the materialized view to reduce IO, and view maintenance in the face of updates.

Three types of view implementations are possible [26]. The *fully materialized* view stores all relevant information, i.e. all shortest paths, the cost of the shortest paths, or both are pre-computed. A *virtual view* relies on on-line computation and stores neither the shortest paths nor their cost. The *hybrid view* stores part of the relevant information, for example, storing the costs of the shortest paths but not the shortest path itself.

This paper explores hybrid materialization to analyze the storage/computation-time trade-offs in materializing individual views needed for processing shortest-path queries. Our experiments with the Twin Cities metropolitan road-map show that materializing the shortest-path-cost view for the boundary graph provides the best savings in computation time for a given amount of storage when the number of fragments is small. Materializing the relevant part of the shortest-path-cost view for the fragment graphs provides the next-best savings, followed by materializing the shortest-path view for the boundary graph. Virtual shortest-path view

for fragments can reduce the storage cost by an order of magnitude or more for large graphs.

1.3 Scope of Paper and Outline

This paper focuses on materialization issues primarily and makes simple choices for resolving other issues. For this study, we work with a hierarchical algorithm with a two level hierarchy. The issue of view maintenance in the face of updates, buffering in DRDB, etc. will be addressed in future work. Min-cut partitioning [14] is used to decompose the base graph into fragments. The shortest-path-cost view and the shortest-path view are clustered by the source-node. The design decisions made to resolve these issues are orthogonal to the issue of materialization and have been studied in the literature.

Section 2 describes the basic concepts in hierarchical routing, giving a description of the algorithm and a proof of correctness. Section 3 introduces the concept of and describes the choices in view materialization. In section 4, we present experimental results for the materializations in terms of CPU operations, IOs required, MMDB size, and storage space used. Section 5 summarizes our work and presents future challenges in hierarchical routing.

2 Basic Concepts

2.1 Flat-Graph Definitions

Graph $G = (N, E, C)$ is a flat graph consisting of a node set N, a cost set C, and an edge set E. The edge set E is a subset of the cross product $N \times N$. Each element (i, j) in E is an edge that joins node i to node j. Each edge (i, j) is associated with a cost $c_{i,j}$. Cost $c_{i,j}$ takes values from the set of positive real numbers. A node i is a neighbor of node j if $E_{i,j} \in E$. The degree of a node is the number of neighboring nodes. A path $P_{s,d}$, in a graph from a source node s to a destination node d, is a sequence of nodes (N_0, N_1, N_2, ..., N_k), where s = N_0, d = N_k, and the edges $E_{0,1}, E_{1,2}, ..., E_{k-1,k} \in E$. The cost of the path is the sum of the cost of the edges, i.e. $\sum_{i=1}^{k} c_{i-1,i}$. An optimal path from node i to node j is the path with the smallest cost.

2.2 Hierarchical Graph Definitions

The hierarchical graph has a 2-level representation of the original graph. The lower level is composed of a set of fragments of the original graph. The higher-level graph is comprised of the boundary nodes and is called the boundary graph. Boundary nodes are defined as the set of nodes that have a neighbor in more than one fragment, i.e. $N_i \in BN \iff \exists E_{i,j}, E_{i,k} | FRAG(k) \neq FRAG(j)$. Edges in the boundary graph are called boundary edges, and the boundary nodes of a fragment form a clique, i.e. are completely connected. The cost associated with the boundary edge is the shortest-path cost through the fragment between the boundary nodes. A boundary edge is associated with a fragment identifier. A ***boundary path*** is the shortest path through the boundary graph.

Theorem 1 *$BG.SP(s,d)rewrite[Edge \rightarrow] = G.SP(s,d)rewrite$[2] $[BoundaryNode \rightarrow BoundaryNode, InteriorNode \rightarrow, Edge \rightarrow]$ if $s, d \in BG$*

Proof: Consider an optimal path from s to d in G. From this path, drop all the edges and the interior nodes. The resulting path p_g is a path consisting only of boundary nodes. Now, considering any pair of adjacent boundary nodes, both the boundary nodes share a common fragment and its interior nodes; also

[2] Rewrite [9] in this case drops all the interior nodes and edges in the path and keeps just the boundary nodes

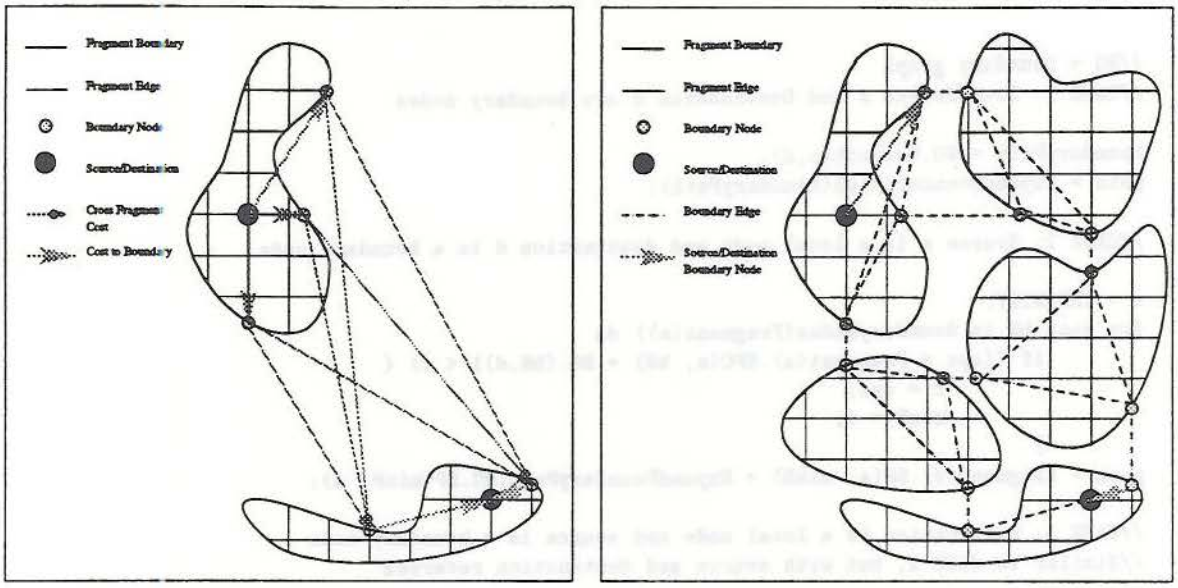
```
1   //BG = boundary graph
2   //CASE 1: Both Source s and Destination d are boundary nodes
3
4   boundaryPath = BG.GetPath(s,d);
5   path = ExpandBoundaryPath(boundaryPath);
6
7   //CASE 2: Source s is a local node and destination d is a boundary node
8
9   c = INFINITY;
10  for each bN in BoundaryNodes(Fragment(s)) do
11          if ((cpc = Fragment(s).SPC(s, bN) + BG.(bN,d)) < c) {
12                  c = cpc;
13                  minB = i;
14          }
15  path = Fragment(s).SP(s, minB) + ExpandBoundaryPath(BG.SP(minB, d);
16
17  //CASE 3: Destination is a local node and source is a boundary node
18  //Similar to CASE 2, but with source and destination reversed
19
20  //CASE 4: Both source and destination are local nodes
21  c = INFINITY;
22  for each sBN in BoundaryNodes(Fragment(s)) do
23          for each dBN in BoundaryNodes(Fragment(d)) do
24                  if ((cpc=Fragment(s).SPC(s,sBN) + BG.SPC(sBN,dBN)
25                       + Fragment(d).SPC(dBN,d))< c) {
26                                  c = cpc;
27                                  minSB = sBN;
28                                  minDB = dBN;
29                  }
30  if (Fragment(s) == Fragment(d) && c > Fragment(s).SPC(s,d))
31          path = Fragment(s).SP(s,d);
32  else
33          path = Fragment(s).SP(s, minSB) + ExpandBoundaryPath(BG.SP(minSB, minDB))
34          + Fragment(d).SP(minDB, d);
```

Figure 2: Hierarchical Routing Algorithm Template

the path between them is optimal in that fragment, because the path from s to d is optimal in G. Hence, there will exist a boundary edge in the boundary graph between these boundary nodes corresponding to the shortest path in that fragment, thus p_g exists in the boundary graph. Also, since all the edges in the boundary graph correspond to a shortest path in their individual fragments, there does not exist any path shorter than p_g in the boundary graph between s and d, therefore p_g is the shortest directed path from s to d in BG. This can be proved by induction on the number of boundary edges in the shortest path through the boundary graph□

Theorem 2 $p_g = (G.SP(s,d)rewrite[BoundaryNode \rightarrow BoundaryNode, InteriorNode \rightarrow, Edge \rightarrow]) \in P_{set}$.
where $P_{set} = \{p_{ij} rewrite[Edge \rightarrow] \mid p_{ij} = BG.SP(b_i, b_j), b_i \in BoundaryNodes(Fragment(s)),$
$b_j \in BoundaryNodes(Fragment(d))\}$

Proof: Consider an optimal path from s to d in G. The sub-path between the first boundary node b_i and the last boundary node b_j in the path is optimal in G, hence, from *Theorem 1*, we have $p_g \in P_{set}$. □



(a) Step 1: Finding the Cost from Source/Destination to their Boundary Nodes

(b) Cross Fragment Costs are used to Determine which Boundary Nodes Must Be on the Path

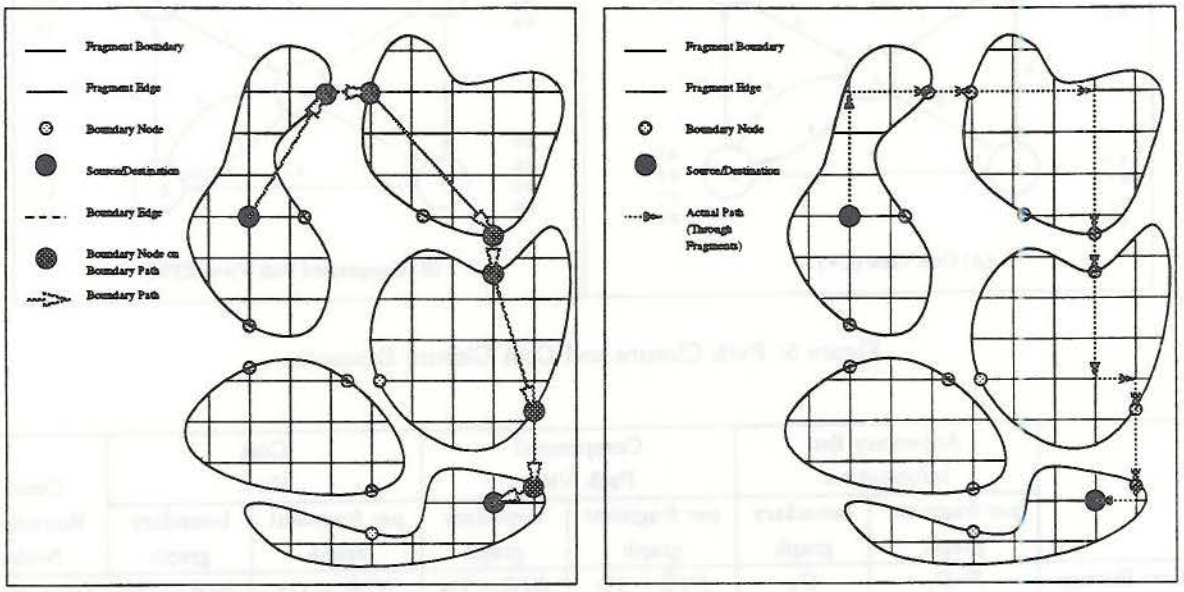
Figure 3: Routing Example

2.3 Hierarchical Routing Algorithm

Figure 2 gives a template algorithm for finding a path in a hierarchical graph. The following queries are sent to the database to compute the optimal path:

- **SP(s,d)** : This query returns the shortest path between node s and d.
- **SPC(s,d)** : Query to find the cost of the shortest path from a source s to a destination d.
- **BoundaryNodes(f)** : Query that returns the set of boundary nodes for fragment f.
- **Fragment(n)** : This query returns the fragment identifier for interior node n.
- **ExpandBoundaryPath(boundaryPath)** : Query that expands the path through the boundary graph and returns the corresponding path in G by using the **ExpandBoundaryEdge(boundaryEdge)** query for each edge in the boundary path.
- **ExpandBoundaryEdge(boundaryEdge)** : This query expands a boundary edge and returns a corresponding path in G by computing the shortest path between the endpoints of the edge in the fragment.

The hierarchical algorithm is composed of three steps: finding the relevant boundary-node pair in the boundary graph, computing the boundary path, and expanding the boundary path. The first step in determining the shortest path is to compute the boundary node through which the shortest path leaves the source's fragment and enters the destination's fragment. If both the source and destination are boundary nodes, then it is trivial. If the source is an internal node and the destination is a boundary node, the boundary node through which the shortest path leaves the source's fragment is found by querying the fragment



(a) Step 3:Finding the Path Through the Boundary Graph

(b) Step 4: Finding the Shortest Path using Boundary Path Information

Figure 4: Routing Example

graph for the cost of the path from the source to all boundary nodes of that fragment, and by querying the boundary graph for the cost of the shortest path from all boundary nodes of the source's fragment to the destination. The source-boundary-destination triple with the lowest aggregate cost determines the appropriate boundary node. The case where the source is a boundary node and the destination is an internal node is similar, but the roles of the source and destination are reversed. When both the source and destination are internal nodes, the appropriate boundary node pair is found by querying the fragment graphs to determine the cost of the shortest path from the internal nodes to all boundary nodes of the fragment, as shown in Figure 3(a). Next, the boundary graph is queried to compute the shortest path cost between all pairs of boundary nodes, as illustrated in Figure 3(b). The path with the lowest aggregate cost determines the boundary-node pair. Once the appropriate boundary-node pair has been determined, the boundary graph is queried to determine the shortest path between those boundary nodes, as Figure 4(a) shows. The final step is to expand the boundary path by querying the fragments for the shortest path through them. Adjacent nodes in the boundary path form source/destination pairs on which the shortest-path query can be run on in a fragment as shown in Figure4(b).

Lemma 1 *The hierarchical routing algorithm finds an optimal path from s to d.*

Proof: The shortest path (s,d) is either interior to a fragment or has at least one boundary node. The former case is covered by calling frag[i].GetPath(s,d). It is important to note that even if the source and destination are in the same fragment, the shortest path between those two points may leave the fragment. In the latter case, the shortest path (s,d) can be broken into three components: The path from s to the first boundary node b_1, then the path from b_1 to the last boundary node b_l, and finally the path from b_l to d. From *Theorem 2*, the path from b_1 to b_l is in P_{set}. □

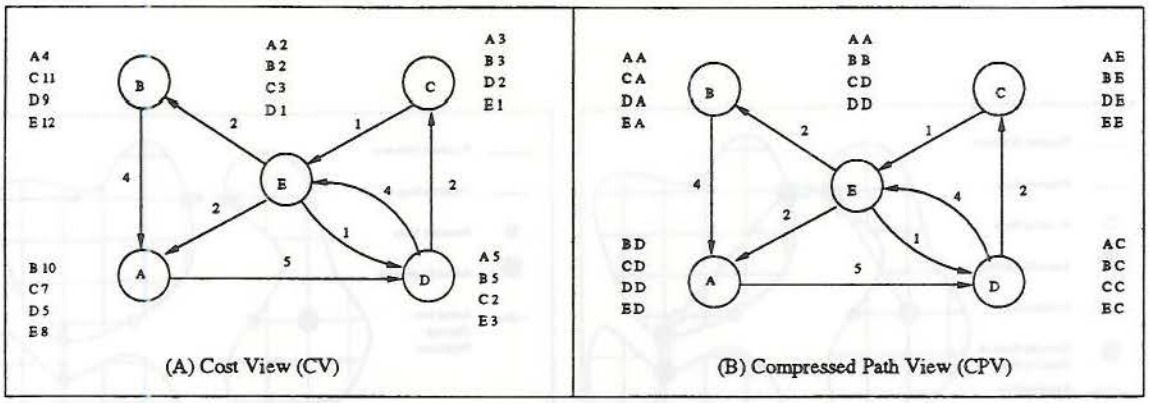


Figure 5: Path Closure and Cost Closure Examples

	Adjacency list information		Compressed Path View		Cost View		Cost2-Boundary-Nodes
	per fragment graph	boundary graph	per fragment graph	boundary graph	per fragment graph	boundary graph	
Raw Storage	$F*G_f$	G_b	$N(\frac{N}{F}-1)*2S_{id}$	$B(B-1)*3S_{id}$	$N(\frac{N}{F}-1)*(S_c+S_{id})$	$B(B-1)*(S_c+S_{id})$	$(N-B)K*(S_c+S_{id})$
Asymptotic Storage	$O(N)$	$O(N)$	$O(N^{5/3})$	$O(N^{4/3})$	$O(N^{5/3})$	$O(N^{4/3})$	$O(N^{4/3})$
Real Size F=100	4M	3M	2.3G	32M	2.3G	32M	40M

Table 1: Asymptotic Size and Complexity of the Materialized Views

3 Degree of Materialization in Hierarchical Routing Algorithms

3.1 View Materialization Terminology

A cost view (CV) on a graph materializes the cost of the shortest path between all node pairs in the graph. It does not store any path information. A compressed path view CPV stores the set of optimal paths between all points on the graph as a series of "hops". For a fragment graph, a partial materialization of the CV, the $C2B$ or cost-to-boundary-nodes view stores the cost of the shortest path from the interior nodes of the fragment to the boundary nodes of the fragment.

Figure 5 shows CPV and CV for an example graph with 5 nodes and 8 edges. Each node is associated with a table, with one entry for all other nodes. The node table in CV contains the costs of the shortest paths between two nodes. For example, the cost from node C to node A is given as 5. To find the shortest path between those two nodes, the CPV table is used. Starting at the table for node C, the destination is looked up to get the next node on the path. In this case, the next destination is node E, and the CPV table for node E is used. From that table, node A is found to be the next hop and the destination is reached.

Symbol	Explanation	Symbol	Explanation
N	Number of nodes	S_c	size of storing a cost
S_{id}	Size of storing a node identifier	B	Number of boundary nodes
F	Number of fragments	G_b	Size of boundary graph in bytes
K	Average number of border nodes per fragment	L	Length of path through base graph
M	Length of path through boundary graph		

Table 2: Table of Notations

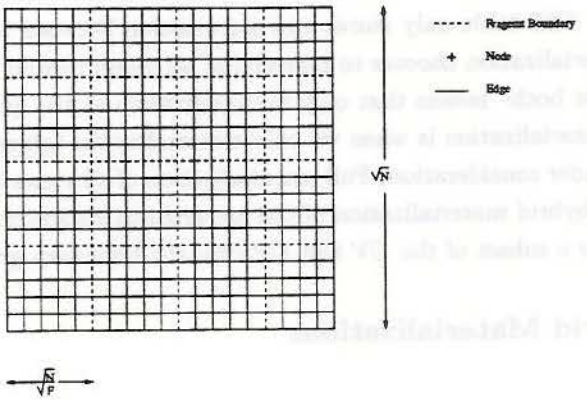


Figure 6: Square Grid Graph

3.2 Asymptotic Analysis of Storage Cost Using a Synthetic Grid Graph

A summary [3] of the parameters used in the remainder of the paper is given in Table 2. A square grid-graph of N nodes is an undirected planar graph. If $\sqrt{N}$ is an integer, the square grid-graph can be embedded in a Cartesian space with nodes at coordinates $(i*\sqrt{N}, j*\sqrt{N})$ with $i, j \in 0, 1, 2, ..., \sqrt{N}-1$. Edges connect nodes $(i\sqrt{N}, j\sqrt{N})$ to nodes $((i+1)\sqrt{N}, j\sqrt{N})$ and $(i\sqrt{N}, (j+1)\sqrt{N})$. A grid-graph is often used in complexity analysis for transportation algorithms because of its regular structure and similarity to urban road-maps [21].

In a graph decomposed into F fragments, each fragment has approximately N/F nodes. The average number of boundary nodes for each fragment is $K = 4\sqrt{N/F}$. The total number of boundary nodes is approximately $B = \sqrt{NF}$. Table 1 shows the view storage requirements. The storage formulas for the materialized views are shown first and are given in terms of N, B, K, and constants. F can be chosen to minimize the storage of the materialized views, as is studied in [24]. Setting F in terms of N, the asymptotic storage sizes for the views are computed. When $F = N^{1/3}$, the storage that minimizes the size of CV and CPV for the boundary graph, the values $K = N^{1/3}$ and $B = N^{2/3}$ are used to compute the asymptotic storage sizes for the views. The asymptotic analysis shows that the sizes of the CPV and CV views for the fragments is $O(N^{1.66})$, which is larger than the CPV and CV for the boundary graph and $C2B$ for the fragments which grow at $O(N^{1.33})$. This is verified in the real sizes of the views are given for $F = 100$ ($F \approx N^{2/5}$). The CPV and CV views for the fragments take a couple of gigabytes of storage, while the other views are two orders of magnitude smaller, as shown in Table 1. The will not consider the materialization of the CPV on the fragments further in this paper. The size of the CPV for the base graph is $O(N^2)$, which

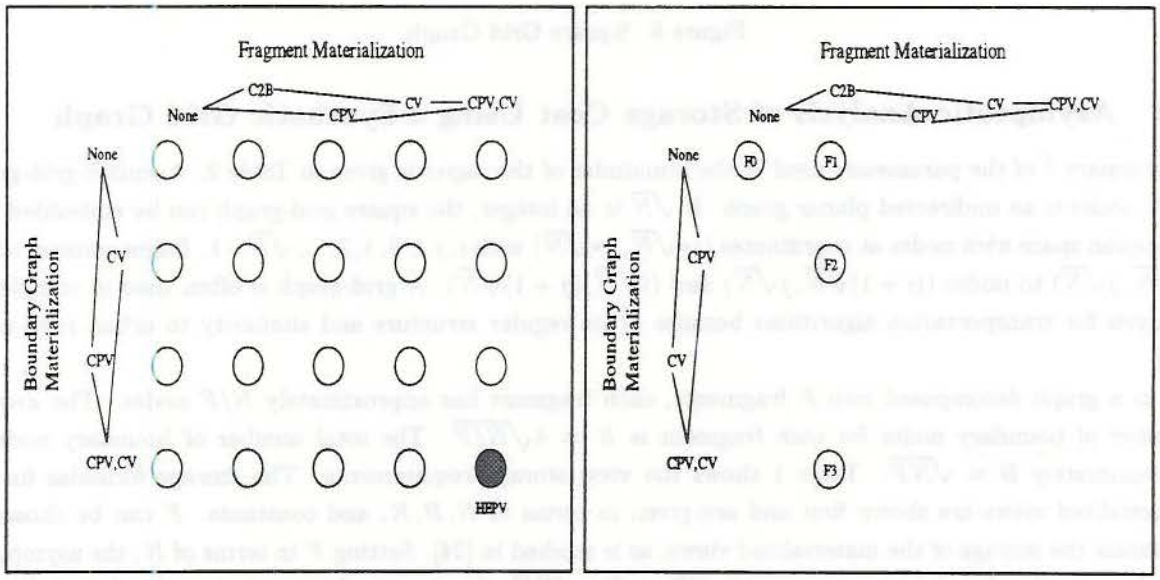
[3] The details of the analysis are available in Appendix A to facilitate the review process. The appendix would be available to readers on a web site, but not included in the final paper if space is at a premium.

would take about 100G to store.

3.3 Levels of Hybrid Materialization

A fully materialized view has all relevant information pre-computed and stored, while a virtual view has no information pre-computed. Hybrid materialization forms the middle ground between those two approaches – some of the relevant data is materialized, while some is left for computation. In terms of the hierarchical routing algorithm, three different levels of hybrid materialization could be used: table level, graph level, and algorithm level. Hybrid materialization with respect to a single table means that only part of the table is kept as a materialized view. An example of this is the *C2B* view which materializes only a part of the *CV* table for the fragment graphs. In this case, the *CV* table is fully materialized because it stores cost information between all pairs, while the *C2B* table only stores cost information between internal nodes and boundary nodes. The graph-level materialization chooses to materialize all views related to either boundary graph or the fragment graphs, but not both. means that only materialization at the graph level is considered. The third definition of hybrid materialization is when we only materialize the tables necessary for the version of the hierarchical algorithm under consideration. Full materialization of all views for the hierarchical algorithm was researched in [13]. The hybrid materialization of the hierarchical algorithm (the focus of the remainder of the paper) will materialize a subset of the *CV* and *CPV* for the boundary graph and fragment graphs.

3.4 Candidate Hybrid Materializations



(a) Materialization Matrix

(b) Candidate Hybrid Materializations for Experiments

Figure 7: Degree of Materialization in Hierarchical Routing Algorithms

Figure 7(a) shows the possible hybrid materializations for the hierarchical algorithm. Materialization choices may be made for both the boundary graph and the fragment graph. The horizontal axis refers to the materialization possible in the fragmented graph, and the vertical axis refers to the materialization possible

for the boundary graph. The axis labels are in lattice form to indicate that a partial ordering exists among the materialization choices. The first choice in materialization for the fragment graph is to not materialize any views (column 1 in Figure 7(a)). Thus, all computation is performed on-line. The next set of choices involve materializing only a single view (columns 2-4 in Figure 7(a)) in the fragment graph, the *C2B* and *CPV* tables. If the *C2B* view is materialized, then the calls to frag[i].SPC(in,bn) in the algorithm template correspond to lookups instead of to on-line computation. The *C2B* view is a subset of the *CV* view for the fragment graph, but materializing the full *CV* is not interesting since the algorithm does not use any of the extra information. If the *CPV* is materialized, computing the shortest path inside of a fragment corresponds to a series of lookups instead of to on-line computation. It is also possible to materialize both the *CV* and *CPV* of the fragment graph (column 5 in Figure 7A) so that no computation is done on-line. Materialization choices for the boundary graph proceed similarly. The boundary graph can have no materialization (row 1 in Figure 7A). If this is the case, all computation in the boundary graph is done on-line. Likewise, any of the views for the boundary graph can be materialized independently (rows 2-3 in Figure 7A). Also, all boundary-graph views can be materialized (row 4 in Figure 7A). Thus, the overall degree of hybrid materialization for the algorithm is defined as the amount of materialization in the boundary graph and fragment graph. Hybrid materializations can exist in any circle in Figure 7(a). For our experiments, we focus on the hybrid materializations shown in Figure 7(b).

4 Experiment Evaluation of Hybrid Materialization

4.1 Experiment Design

4.1.1 Goals

The goal of the paper is to study the effects of materializing individual views on fragments and on the boundary graph on the storage and computation cost of the hierarchical routing algorithm. Computation cost is measured by the total number of IO's required and the on-line computation effort in terms of heap operations and heap size.

The candidate hybrid materializations are evaluated using the experimental setup shown in Figure 8. Benchmark maps are first converted to the input format of the *graph fragmentizer* and the *work load generator*. The fragmented graph is used by the *hierarchical routing application*. The *view manager* is responsible for implementing candidate hybrid materializations. The materialized views are partitioned between the MMDB and DRDB by the *view partitioner*. The *work-load generator* uses the list of nodes to generate (source,destination) pairs. The *hierarchical routing application* executes the hierarchical routing algorithm using the views stored in the MMDB and DRDB with the (source,destination) pairs. Statistics from the *heap manager* and *IO manager* are collected for *data analysis*.

4.1.2 Hybrid Materialization Candidates

We chose four candidate hybrid materializations for direct comparison, as shown in Figure 7(b). These candidates are chosen to facilitate studying the effects of materializing individual views on the fragment graph and the boundary graph. F0 has no materialization in either the boundary graph or the fragment graphs. F1 only materializes the *C2B* table in the fragment graph. F2 materializes the *C2B* table of the fragment graph and the *CV* table of the boundary graph. F3 materializes the *C2B* table in the fragment graph and both the *CPV* and *CV* tables in the boundary graph. We compare the candidates to determine the effect of materialization on the hierarchical algorithm. We compare F0 and F1 to determine the effect

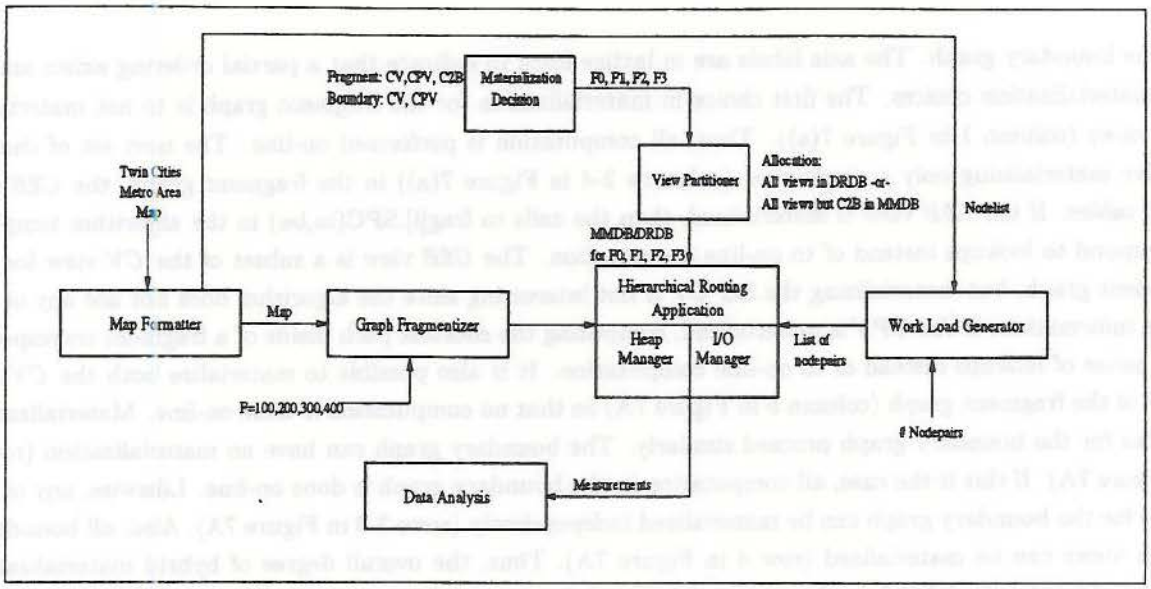


Figure 8: Experimental Method for Evaluating Shortest Path Algorithms

	Adjacency list information		Compressed Path View		Cost View		Cost2-Boundary-Nodes
	per fragment graph	boundary graph	per fragment graph	boundary graph	per fragment graph	boundary graph	
F0	$2K + M + 2$	$K^2 + 1$					
F1	$M + 2$	$K^2 + 1$					2
F2	$M + 2$	1				K	2
F3	$M + 2$			M		K	2

Table 3: Views and their Access Patterns for Various Algorithms

of materializing the *C2B* table in the fragment graph. By comparing F1 and F2, we determine the effect of materializing the *CV* in the boundary graph. Comparing F2 and F3, we determine the effect of materializing the *CPV* view in the boundary graph.

Table 3 shows the patterns of access to the views for the different candidate materializations we have chosen to study. The table is summarized below.

- **F0:** There will be K^2 A* run in the boundary graph to compute the cross-fragment paths. In the fragment graph, there are $2K$ A* run to determine the cost to the boundary nodes, $M + 2$ A* to expand the boundary path, and 1 A* in the boundary graph to compute the boundary path
- **F1:**This is identical to F0, but the $2K$ A* are replaced with 2 lookups to the *C2B* view.
- **F2:** F2 is identical to F1, but the K^2 A* are replaced with K lookups in the *CV* for the boundary graph.
- **F3:** This case is identical to F2, except that the boundary path is determined using the *CPV* instead of running an A*.

4.1.3 Partitioning of Views Between MMDB and DRDB

In the experiments, we look at two partitioning strategies. In the first partitioning, the *C2B* table was stored in the DRDB and the rest of the views were stored in the MMDB. In the other partitioning, all views are stored in the DRDB with only the adjacency-list of the fragment graphs and border graph are in main memory.

4.1.4 Metrics

The storage cost is measured by the number of bytes needed to store the data-structures i.e. graphs, views. The computation time has two components: i) CPU time and ii) disk IO time. CPU time is determined by the size of the graph and the number of A*s run. One measure of CPU time is the total number of heap operations used by the A* algorithm and the size of the heap when those operations are made. Those parameters give an indication of how many operations need be done and how expensive those operations are. IO cost is dominated by the number of disk IOs incurred to fetch data which cannot be loaded into the MMDB from the views.

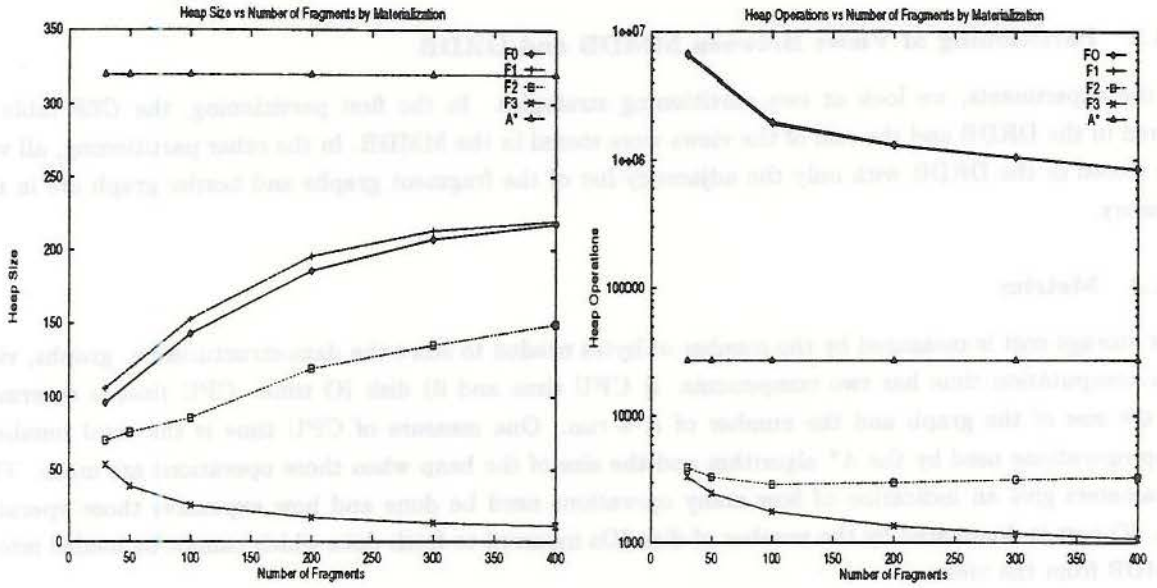
4.2 Comparing Hybrid Materialization Candidates

We report the CPU-costs, IO-costs, and storage-cost of the candidate hybrid materialization strategies in our experiments using the Twin Cities metropolitan road map with 123,000 nodes and 313,000 edges. The are analyzed to deduce the effect of materializing individual views on the performance of the hierarchical routing algorithm.

4.2.1 CPU Cost Comparison

We measure the average heap size and average number of heap operations for the hierarchical routing algorithm using alternative materializations. A set of 500 randomly chosen (source,destination) pairs are used for the experiments. The number of heap operations include inserts and deletes only. A count of these operations is measured for each path and an average is reported in Figure 9(b). For each path, the heap-manager measures the heap-size at every operation (insert/delete) and produces and average heap size for a given (source,destination) pair. We report the average of these over 500 paths in Figure 9(b).

We compare the heap results against running A* on the base graph to get a baseline performance measurement. The effect of creating a hierarchy of graphs is apparent, as all hierarchical algorithms have a smaller average heap size than when running A* on the base graph, as shown in Figure 9(a). A* has a large average heap size since it searches the entire breadth of the graph, while the hierarchical algorithms are focused on the fragments or boundary graph. If the *C2B* table is materialized as in F1, the average heap size increases, since there are $2K$ fewer A* run in the fragment graphs, but the same number of A* run in the boundary graph. The effect of materializing the *CV* in the boundary graph is apparent in F2. In this case, the average heap size decreases dramatically, since the K^2 A* in the boundary graph are replaced with lookups. The cost of running A* in the boundary graph is further highlighted by materializing the *CPV* for the boundary graph (F3), which replaces the A* and finds the path through the boundary graph with a lookup. The average heap size is small, because the only A* are run in the fragments. The average heap size increases as the number of fragments increases when there is no materialization, or when the *C2B* and *CV* are materialized, since the number of nodes in the boundary graph grows. When the *CPV* in the boundary graph is materialized, the average heap size decreases, because no A* are run in the boundary graph and the size of the fragments decreases.



(a) Average Heap Size per Heap Operation vs. Number of Fragments

(b) Average Number of Heap Operations per Path vs Number of Fragments

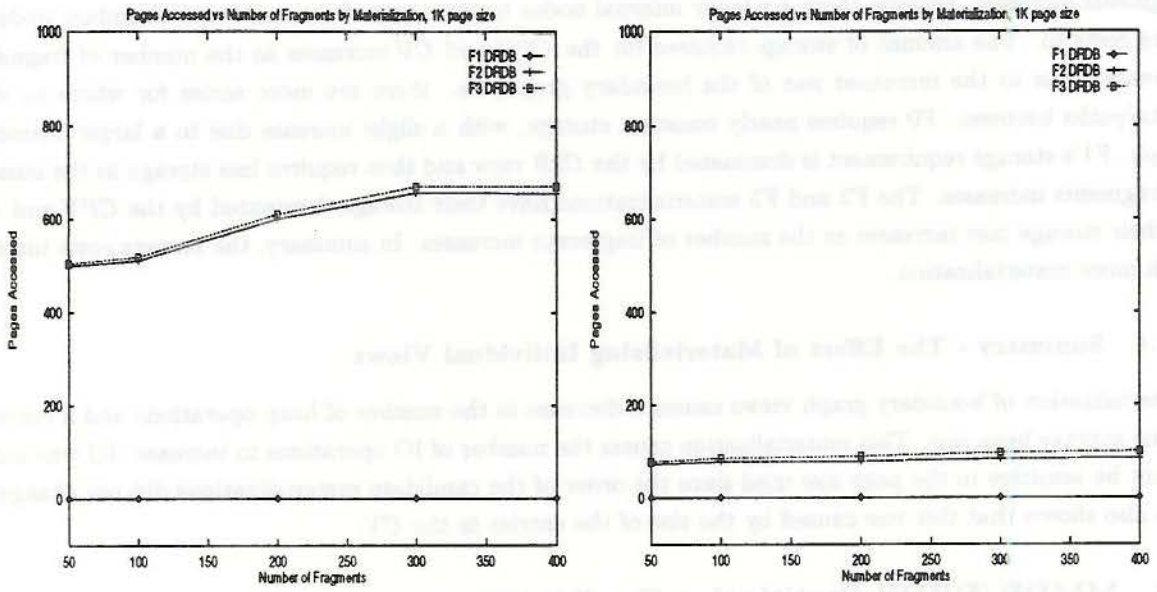
Figure 9: CPU Cost vs Number of Fragments

The number of heap operations is a count of the number of inserts into and deletes from the heap data-structure and is shown in Figure 9(b). Materializing the hierarchical graph (F0) causes the number of heap operations to increase, as compared to running A* on the base graph due to running the K^2 A* in the boundary graph. Materializing the *C2B* view (F1) decreases the number of heap operations slightly, but the K^2 factor still dominates. The hierarchical algorithm will have fewer heap operations than A* only when the CV is materialized (F2), since the K^2 boundary graph A* are replaced with lookups. Materializing the the CPV causes the heap operations to be minimized, since all computation is done in the fragment graphs. The number of fragments inversely affects the number of heap operations for two reasons. With a greater number of fragments, the number of boundary nodes per fragment decreases, lowering the K^2 term. Also, more fragments means that individual fragments will be smaller, thus requiring fewer heap operations to search them. In summary, the number of heap operations decreases as more views are materialized. since it is exploited by F1-F3 directly.

4.2.2 IO Cost Comparison

Clustering Views Clustering the views improves IO performance in the DRDB since the clustering is directly exploited by F1-F3. The *C2B* view is clustered around an internal node. Thus, one lookup to the clustered table retrieves the cost to all boundary nodes, as opposed the the K lookups (one for each boundary node in the fragment) the unclustered view would need. The CV is clustered around fragments – each boundary node stores the cost to all the boundary nodes in other fragments consecutively, resulting in K lookups instead of K^2 lookups.

Logical Accesses We measured logical IOs to simplify interpretation and to avoid platform specific noise. The experiment setup counts the number of access to the views on disk and converts that number to



(a) 1KB Page Size

(b) 8KB Page Size

Figure 10: IO Cost Per Path vs Number of Fragments

the equivalent number of pages using the record page-size. We count each page to be a distinct IO. The experiment does not capture the effect of DRDB-buffering, page clustering, and related optimizations not apparent at the interface level.

Figure 10 shows how the number of accesses varies for the different hybrid materializations considered. F0 has no IO. In F1, the only IO comes from accessing the *C2B* view, which takes at most two accesses. F2 materializes the *CV* for the boundary graph and thus accesses more pages than F1. The difference in the amount of IO between F2 and F3 is the length of the path through the boundary graph. Increasing the number of fragments increases the number of boundary nodes, which has two effects on IO performance. Since the *CV* is clustered, an increase in the number of boundary nodes causes the size of each entry in that table to grow from 14K with 50 fragments to 47K with 400 fragments. The size of the entries is apparent in Figure 10(b) as the order of the materializations did not change as the page size grew. The second increase in IO comes from the increases length of the boundary path.

4.2.3 Storage Costs

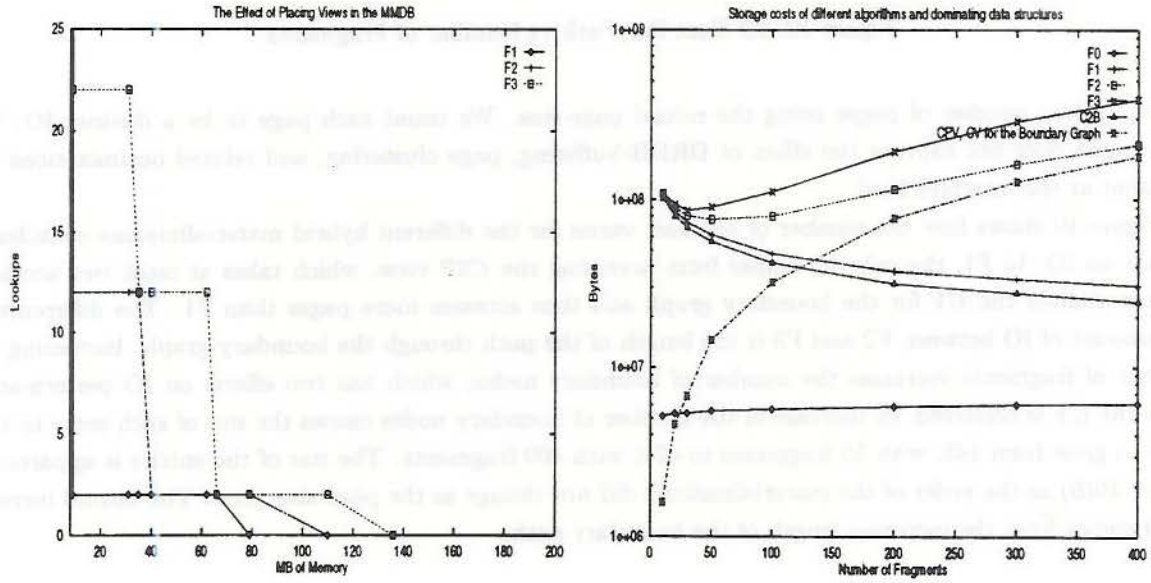
Storage costs were attained by creating the necessary views for the candidate hybrid materializations. The graph was fragmented using a min-cut [14] technique with values of F ranging from 10 to 400. Notations are summarized in Table 2. All candidates store the base and fragment graphs, whose storage requirements are similar to those of the base graph. The storage for the *C2B* view is $(N-B)K(S_c+S_{id})$ bytes. Each of the $(N-B)$ internal nodes have information for the cost to the border node for each of the K boundary nodes of their fragment. The *CPV* requires $B(B-1)3S_{id}$ bytes of storage. Each of the B boundary nodes stores the boundary node identifier, and the source and destination identifiers for the $(B-1)$ boundary nodes. Storing the *CV* takes $B(B-1)(S_c+S_{id})$ bytes. Each of the B boundary nodes stored the cost to the $(B-1)$ other boundary nodes. On our system, S_{id} was 4 bytes and S_c was 8 bytes. Storage was measured using the UNIX 'ls -l' command.

Figure 11(b) shows the storage requirements for the candidate hybrid materializations and for the views that compose those materializations. The storage required for the *C2B* view decreases as the number of fragments increases, because there are fewer internal nodes to store costs from and fewer boundary nodes to store costs to. The amount of storage required for the *CPV* and *CV* increases as the number of fragments increases, due to the increased size of the boundary graph, i.e. there are more nodes for which to store costs/paths between. F0 requires nearly constant storage, with a slight increase due to a larger boundary graph. F1's storage requirement is dominated by the *C2B* view and thus requires less storage as the number of fragments increases. The F2 and F3 materializations have their storage dominated by the *CPV* and *CV*, so their storage cost increases as the number of fragments increases. In summary, the storage costs increase with more materialization.

4.2.4 Summary - The Effect of Materializing Individual Views

Materialization of boundary graph views causes a decrease in the number of heap operations and a decrease in the average heap size. This materialization causes the number of IO operations to increase. IO was shown to not be sensitive to the page size used since the order of the candidate materializations did not change. It was also shown that this was caused by the size of the entries in the *CV*.

4.3 MMDB/DRDB Partitioning Candidates



(a) Partitioning Views, F=100,M=10,L=350

(b) Storage Cost Requirements For Various Materializations

Figure 11: View Partitioning and Storage Requirements

Partitioning views in MMDB separates the data stored in the MMDB from the data stored in the DRDB. From Figures 10(a)(b) and 9(a)(b), the most costly operation of the algorithm is accessing the boundary-graph cost view. If that view is virtually materialized, then the CPU cost becomes prohibitive, while if the view is fully materialized but in a DRDB, the IO is costly. Thus, the *CV* becomes the first candidate for

placement in the MMDB. The next view to be placed in the MMDB is the *CPV*, since it is accessed M times, where M is the length of the boundary path. The last view to be placed in the MMDB should be the *C2B* table, since it is accessed twice at most.

Figure 11(a) shows the effect view-partitioning has on the number of disk lookups for each candidate hybrid materialization. This figure assume that the graph has 123,000 nodes, that the path length is 350, and that the boundary path length is 14 (variables characteristic a long path in the graph). For this experiment, a disk lookup is an access to the disk, not a count of the number of pages read. The first area of transitions occurs when enough memory becomes available to place the fragment graphs and boundary graphs in the MMDB. The next important transitions occur in the 30-40M range. At this time, enough memory is available for the *CV* to be in the MMDB. With around 70-90M of memory, the *C2B* can be placed in the MMDB, which eliminates IO for F1 and F2. IO for F3 is eliminated when enough memory becomes available for all three views, which occurs around 110M.

5 Conclusion

In this paper, we have addressed two important issues regarding the hierarchical routing algorithm. The first issue was to determine which views to materialize if storage is limited. We found that materializing the shortest-path-cost view for the boundary graph provides the greatest computational savings, followed by materializing the cost-to-boundary-nodes view for the fragments and then the shortest-path view for the boundary graph. Once the appropriate views are materialized, it is necessary to partition the views between the MMDB and the DRDB when the size of the MMDB is limited. We found that accessing the shortest-path-cost view incurs a high IO cost and should be placed in the MMDB first, followed by the shortest-path view for the boundary graph and then the cost-to-boundary-nodes view.

In our future work, we plan on addressing clustering in DRDB, view maintenance and updates, graph fragmentation, and multi-level hierarchies. We also plan to implement a parallel version of the hierarchical routing algorithm.

We would like to thank Rajat Agarwal for his assistance with graph partitioning; Siva Ravada and Xuan Liu for their technical editing, and Christiane McCarthy for improving the overall readability of the paper.

References

[1] R. Agrawal. "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries". *IEEE Trans. on Software Engineering*, 14(7), 1988.

[2] R. Agrawal, A. Borgida, and H.V. Jagadish. "Efficient Management of Transitive Relationships in Large Data Bases". In *Proc. of SIGMOD Intl Conference on Management of Data*. ACM, 1989.

[3] R. Agrawal and H.V. Jagadish. "Hybrid Transitive Closure Algorithms". In *Proc. of Intl Conference on Very Large Data Bases*, 1990.

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

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

[6] Greg Frederickson. "Searching Among Intervals and Compact Routing Tables". *Algorithmica*, pages 448–466, 1996.

[7] Greg Frederickson and Ravi Janardan. "Designing Networks with Compact Routing Tables". *Algorithmica*, pages 171–190, 1988.

[8] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[9] R. H. Guting. GraphDB: Modeling and Querying Graphs in Databases. In *Proc. of Intl Conference on Very Large Data Bases*, 1994.

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