# Efficient Query Processing on Unstructured Tetrahedral Meshes

Stratos Papadomanolakis Anastassia Ailamaki
Julio C. Lopez Tiankai Tu David R. O'Hallaron
Carnegie Mellon University
Pittsburgh, PA 15213
{stratos, natassa, jclopez, tutk, droh}@cs.cmu.edu

Gerd Heber
Cornell Theory Center
638 Rhodes Hall
Ithaca, NY 14853
heber@tc.cornell.edu

## ABSTRACT

Modern scientific applications consume massive volumes of data produced by computer simulations. Such applications require new data management capabilities in order to scale to terabyte-scale data volumes [32, 10]. The most common way to discretize the application domain is to decompose it into pyramids, forming an unstructured tetrahedral mesh. Modern simulations generate meshes of high resolution and precision, to be queried by a visualization or analysis tool. Tetrahedral meshes are extremely flexible and therefore vital to accurately model complex geometries, but also are difficult to index. To reduce query execution time, applications either use only subsets of the data or rely on different (less flexible) structures, thereby trading accuracy for speed.

This paper presents efficient indexing techniques for generic spatial queries on tetrahedral meshes. Because the prevailing multidimensional indexing techniques attempt to approximate the tetrahedra using simpler shapes (rectangles) query performance deteriorates significantly as a function of the mesh's geometric complexity. We develop Directed Local Search (DLS), an efficient indexing algorithm based on mesh *topology* information that is practically insensitive to the geometric properties of meshes. We show how DLS can be easily and efficiently implemented within modern database systems without requiring new exotic index structures and complex preprocessing. Finally, we present a new data layout approach for tetrahedral mesh datasets that provides better performance compared to the traditional space filling curves. In our PostgreSQL implementation DLS reduces the number of disk page accesses and the query execution time each by 25% up to a factor of 4.

## 1. INTRODUCTION

Simulations are crucial for studying complex natural phenomena, from the flow of hot gas inside a propellant to the propagation of cracks inside materials, earthquakes and climate evolution. Recent advances in modern hardware allow scientists to carry out simulations of unprecedented resolution and scale. Accurate simulations improve our understanding and intuition about complex physical processes, but in order to reap their benefits we must be able to search for useful information in the haystack of large-scale simulation output datasets.

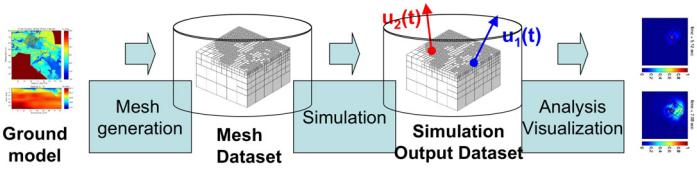


**Figure 1: An earthquake simulation pipeline.**

### 1.1 Querying Simulation Datasets

To analyze and display simulation results, post-processing and visualization applications query a discretized version of the application domain (typically represented using a *mesh*) and the simulation output. Figure 1 shows the architecture of *Hercules*, a simulation application developed by the Quake group at Carnegie Mellon [1, 33, 23], that computes earthquake propagation for a given ground region and initial conditions. The simulation receives as a grid-like discrete ground model and at each simulated time-step it computes the ground velocity at the mesh points, storing the result in the *simulation output*. In *Quake* simulations, mesh models typically consume hundreds of gigabytes and simulation output volumes are in the terabyte scale [1]. To fully utilize the information involved in a modern simulation, post-processing and visualization applications need efficient, scalable query processing capabilities.

To organize the data representing the discretized application domain, simulations typically employ an *unstructured tetrahedral mesh*. A tetrahedral mesh models the problem domain by decomposing it into tetrahedral shapes called *elements*. For instance, Figure 2(a) shows a part of a mechanical component mesh model, whereas Figure 2(b) illustrates a constituent tetrahedral element. The element endpoints, called the *nodes*, are the discrete points on which the simulation computes physical parameter values, like ground velocity. Tetrahedral elements have varying sizes, angles and orientations. When dealing with complex geometries that require variable resolution, the tetrahedral mesh is a powerful modeling tool due to its unique flexibility in defining arbitrarily-shaped elements. Therefore, tetrahedral meshes


*SIGMOD 2006*, June 27–29, 2006, Chicago, Illinois, USA.

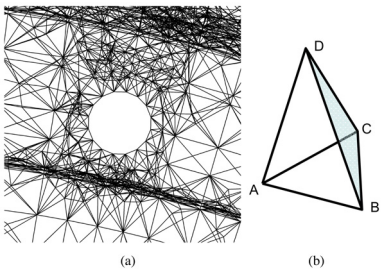


**Figure 2: (a) Part of a tetrahedral mesh dataset modeling a mechanical component. (b) A tetrahedral (pyramid) mesh element and its four endpoints, the nodes.**

are vital for a wide range of applications in areas like mechanical engineering and earthquake modeling.

The most frequent query types on tetrahedral meshes are spatial, point and range queries. They retrieve the mesh elements intersecting an input query region or containing the query points, along with the corresponding mesh nodes. Such queries are important in visualization or analysis applications that interpolate the values of physical parameters (like ground velocity) at a given point or a region, from the values computed at the mesh nodes. We also identify a class of queries called *feature* queries, that retrieve arbitrarily shaped regions of the dataset that are important for the application, such as surfaces and boundaries.

Query processing performance is critical for the scientific processing applications that require interactive rendering rates (less than 1s per frame [35]). High frame rates are impossible to achieve on large-scale mesh datasets without efficient indexing techniques. Unfortunately, the pyramid-based geometry of tetrahedral meshes, while increasing their expressive power, makes developing effective indexing methods a challenging task. To tame the long execution times, scientific applications typically compromise accuracy either by utilizing a subset of the mesh data or by resorting to less flexible structures. Because meshes are difficult to index and query efficiently using spatial indexing methods available in today's Database Management Systems (DBMS), applications use specialized programs instead. As the size and complexity of the datasets grows, however, these programs suffer from scalability, performance, and portability limitations [14].

## 1.2 Previous Work

Current scientific computing and visualization practices rely on storing tetrahedral mesh datasets in main memory, using machines (clusters or supercomputers) with sufficient aggregate memory capacity [24]. This solution becomes impractical for terabyte-scale data volumes (like the 2.5TB model on the Earth Simulator [20]). Furthermore, casting data access as a parallel processing problem introduces unnecessary programming complexity and communication as well as synchronization overheads.

Previous work on *out-of-core* visualization [31] for disk-resident data does not address general-purpose query processing on tetrahedral meshes. The focus is on specialized applications like *particle tracing* [34], that iteratively load parts of the mesh in main memory and process them as a batch. The absence of effective general-purpose indexing for large tetrahedral datasets often forces scientists to use other types of models, like oct-tree meshes, that are easier to index but lack the ability to model geometrically complex problems, like ground or material fractures.

Database literature provides a wealth of spatial indexing techniques [8]. As we show in the next section, existing techniques do not scale when applied on arbitrarily complex tetrahedral meshes, because the pyramids cannot be effectively captured by simple approximations like the Minimum Bounding Rectangle (MBR). Approximations stumble on the irregular pyramid shapes, sizes and angles and incur significant storage overhead and construction costs.

Using DBMS technology to handle the emerging massive datasets in scientific applications is advocated by both senior database researchers and scientists [10]. Commercial DBMS successfully support astronomical databases [17], offering huge benefits in performance and ease of implementation. Commercial DBMS are currently also used to support simulation applications [14]. Our techniques are fully compatible with commercial DBMS technology, utilizing existing access methods and featuring extremely simple pre-processing. Thus our work is one more step towards bridging the gap between scientific applications and modern databases. In fact, our algorithms were recently implemented as a module for SQL Server 2005 and are used for real applications by the Cornell Fracture Group [15]. [1]

## 1.3 Our Approach and Contributions

In this paper, we introduce *Directed Local Search (DLS)*, a query processing approach for tetrahedral mesh datasets. DLS avoids the complexities involved in trying to capture the geometry of the mesh, by utilizing the *connectivity* between mesh elements.

DLS uses a novel application of the Hilbert curve to obtain an initial approximate solution, which is "refined" through local search algorithms. Our technique relies on the distance preserving properties of the Hilbert curve and on an efficient representation of connectivity information to provide significantly better performance compared to traditional techniques that rely only on geometric approximation.

DLS allows the construction of simulation applications that can efficiently query large-scale meshes stored in a database system along with implementation simplicity and easy integration with existing DBMS.

The detailed contributions of this paper are:

1. This is the first paper to treat query processing on tetrahedral mesh data, a crucial problem for large scale scientific applications, using database technology.
2. We evaluate and compare the performance of the prevailing spatial indexing methods when applied on tetrahedral meshes, explaining their inefficiencies.
3. We design Directed Local Search (DLS), an efficient algorithm for indexing and querying large unstructured tetrahedral meshes. To index a mesh efficiently, DLS for the first time:

[1]We are well aware of the current mismatch between scientific applications and databases. We believe it is a matter of definition. The DBMS we refer to in this paper do not need to include features like transactions, which are not required by scientific applications. Instead, we refer to a highly optimized and mature "core" DBMS functionality (like B-Tree indexes and join algorithms) that can benefit scientific data management.

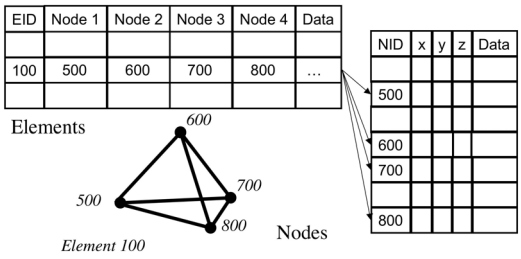


**Figure 3: The mesh representation in the database consists of separate tables for the mesh elements and nodes.**

(a) Combines mesh topology information with the mesh geometry.
(b) Applies the Hilbert space-filling curve for approximate indexing

4. We implement DLS in a simple and efficient fashion, using standard access methods (the ubiquitous B-Tree). In addition, we use graph-based techniques to efficiently store dataset topology information.
5. Experiments with DLS running on top of PostgreSQL show that DLS results in a reduction in the number of I/O accesses and query execution time by 25% up to a factor of 4.
6. We propose a graph-based technique for clustering mesh elements on disk pages and we show that it improves the I/O performance of *feature* queries by 16% to 37.9% compared to traditional linear ordering based on the Hilbert space-filling curves.

This paper is structured as follows. Section 2 details the database design and query workloads. Section 3 evaluates the prevailing spatial indexing techniques on tetrahedral mesh datasets. In Section 4 we describe Directed Local Search and the related indexing and data organization and in Section 5 our element clustering approach. Section 6 details our implementation and experimental setup, while Section 7 presents our experimental results. We conclude with Section 8.

## 2. BACKGROUND

We use the database organization shown in Figure 3. The mesh components, elements (tetrahedra) and nodes (points), are stored in separate tables. Each *Elements* record contains the IDs of the 4 corresponding nodes, while *Nodes* holds the coordinates for each node. This organization is suitable for spatial queries, as it allows fast access to all the nodes of an element. There exist other relational mappings for meshes [14], but they are tuned towards different query types, for instance determining all the elements sharing a given node.

We consider the following 3 types of queries:

1. A point query simply returns the containing element (and its nodes). Post-processing and visualization applications use point queries in order to interpolate the value of a physical parameter on the particular query point, given the values computed by the simulation at the nodes. This general functionality is vital to virtually every application that requires values at points that do not coincide with the input mesh node set.
2. Range queries return a set of elements contained in or overlapping with the rectangular query region, along with the corresponding nodes. A range query is used to retrieve "chunks" of data that are then fed to a (possibly parallel) visualization or analysis tool.
3. *Feature* queries return regions of the dataset that have arbitrary shapes. Features in datasets are defined by the application. For example, in earthquake analysis the ground surface is a feature of particular interest if we want to measure earthquake impact on buildings.

## 3. TRADITIONAL INDEXING ON TETRAHEDRAL MESHES

Database literature provides a wealth of multidimensional indexing techniques. Gaede et al. provide an excellent survey on the topic [8]. In this section we demonstrate that existing techniques have suboptimal performance for tetrahedral meshes and/or exhibit low storage utilization and preprocessing overheads.

R-Tree-based approaches approximate objects by their Minimum Bounding Rectangles (MBRs) and index them with an R-Tree [12] variant. Performance optimizations involve packing, clipping and replicating overlapping MBRs and using more complex bounding shapes (polyhedra). We investigate the applicability of R-Tree based techniques in sections 3.1 and 3.2.

Another approach is to overlay a rectilinear grid over the indexed domain and approximate each object by one or more grid cells. The cells are arranged and indexed using coordinate transformations like the Z-order. Z-order based techniques are investigated in section 3.3.

### 3.1 R-Tree Based Techniques

The R-Tree search for a query starts from the root level and follows a path of internal nodes whose MBR intersects the query range or contains the query point. If multiple nodes at one level match the query criteria the search will follow all possible paths, requiring more page accesses. There exists a large body of research on improving performance by minimizing the area of R-Tree nodes and the overlaps that lead to multiple paths. Dynamic techniques like the original R-Tree construction algorithm [12] and the R*-Tree [3] maintain an optimized tree structure in the presence of data updates. Static techniques like the Hilbert-packed R-Tree [18], the Priority R-Tree [2] and others [28, 5] attempt to compute an optimal R-Tree organization for datasets that do not change.

Tetrahedral meshes are a challenging application for R-Trees because they have many overlapping MBRs. Figure 4 (a) shows a two-dimensional example. The overlap between the R-Tree leaf nodes A and B is significant and it is not clear how to arrange the individual triangles to minimize it. Also, the elongated triangles in the upper right part give a very large surface to node C.

We evaluate R-Tree performance with a real dataset used for crack propagation simulations [14]. We compare two dynamic implementations, the originally proposed quadratic-split R-Tree and the R*-Tree and two packed implementations, the Hilbert-packed R-Tree and the STR-packed R-Tree [21]. The first two are implemented using PostgreSQL, and the others are taken from the "Spatial Index Library" ([13]).

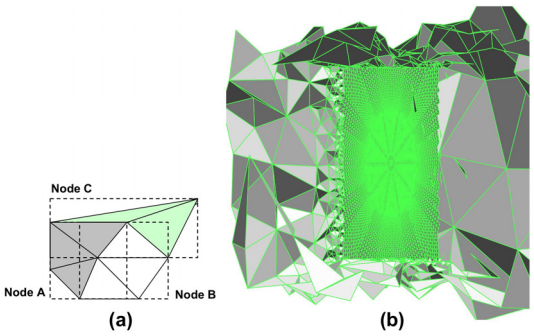


**Figure 4: (a) Overlaps between the leaf-level nodes of an R-Tree for a 2D mesh. (b) The mesh used in the R-Tree experiments.**

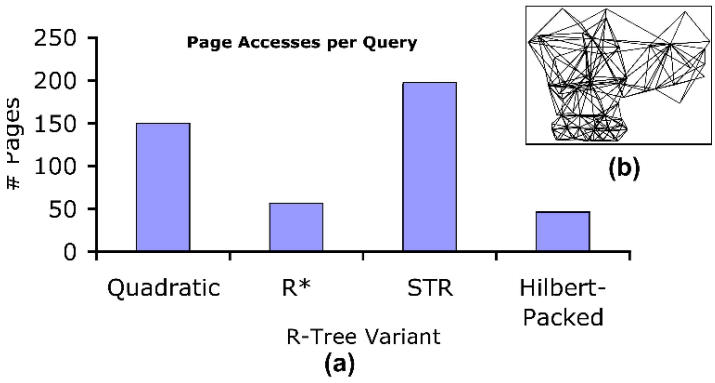


**Figure 5: (a) Average number of page accesses per point query for 4 R-Tree variants, where all trees have 3 levels. (b) Internal structure of a leaf-level node of the Hilbert-packed R-Tree.**

Figure 4 (b) shows part of our dataset structure. Our mesh models a mechanical component with a crack at its center. It contains a very high-resolution central area, surrounded by more coarse-grained elements.

Figure 5 (a) shows the average number of page accesses for the 4 indexes over 1000 point queries focused on the dense region. The indexes perform 30-198 page accesses, while the tree height (and hence the theoretical minimum number of accesses) is 3. Figure 5 illustrates the cause for the measured performance, showing one of the Hilbert-packed R-Tree's leaf nodes. The node has an unnecessarily large volume because of the large elements on the top, that 'cover' the smaller elements. Any query intersecting the bottom right part of the MBR (which is approximately where the dense region of the dataset is) will have to unnecessarily hit that node.

Extensions like the P-Tree [16] attempt to improve R-Tree performance by using Minimum Bounding Polyhedra. The P-Tree relies on polyhedra with faces aligned to a fixed set of $d$ orientations. Such 'constrained' polyhedra will likely still lead to overlaps because they do not capture the geometry of every pyramid in the mesh. In addition, they require additional storage for storing the more complex bounding approximations.

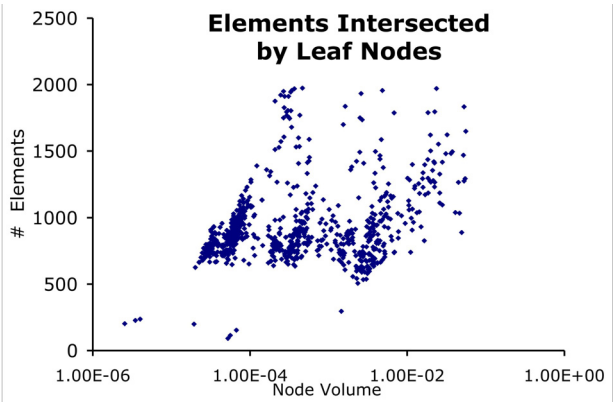


**Figure 6: Number of elements intersected by R-Tree nodes, as a function of node volume.**

## 3.2 Clipping

A clever solution to the problem of overlaps is to use non-overlapping regions. The R+-Tree [29] ensures non-overlapping tree nodes by generating disjoint MBRs during node-splitting and *replicating* the objects that cross MBR boundaries.

The R+-Tree attempts to improve performance by trading search efficiency with higher storage overhead, as pointers to the same object consume space in more than one tree nodes. The increased storage requirements of the R+-Tree make it undesirable for scientific datasets: Simulation datasets are challenging exactly because of their unprecedented volumes and it makes no sense to adopt an indexing solution that multiplies the storage needed for each object!

Storage space is not the only problem: reduced storage utilization negatively affects performance. Unlike point queries, that benefit from the non-overlapping nodes, range queries will suffer because links to objects within the query range will be retrieved multiple times. The performance of loading data in the index will also be problematic, because of the more complicated loading algorithm that also needs to write a lot more data).

We highlight the R+-Tree inefficiency by showing how indexing a tetrahedral mesh dataset requires an unreasonable amount of replication. We measure the number of tetrahedra intersected by leaf-level nodes of different sizes, for the mesh dataset described in Section 3.1, using the nodes generated by the Hilbert R-Tree of Section 3.1 as a guide. Figure 6 shows the number of mesh elements intersected by nodes as a function of the MBR's volume. According to Figure 6, 80% of the nodes intersect 500 to 1000 elements, which will have to be replicated. Given that each node contains 120 entries, the new dataset would require 5 to 10 times more space.

More sophisticated clipping-based techniques, like the disjoint convex polygons of the cell-tree [11] exhibit similar inefficiencies: First, the cell-tree construction algorithm still requires the replication of objects that cross partition boundaries, like in the R+-Tree case. Furthermore, the space overhead of keeping the polygon descriptions in the tree nodes is much higher compared to that of storing MBRs and leads to poor storage utilization for large datasets.

| Grid Resolution | Number of intersected elements |
|---|---|
| 16x16x16 | 100 |
| 32x32x32 | 40 |
| 64x64x64 | 20 |
| 128x128x128 | 10 |

**Table 1: Number of mesh elements intersected by grid cells with varying resolution.**

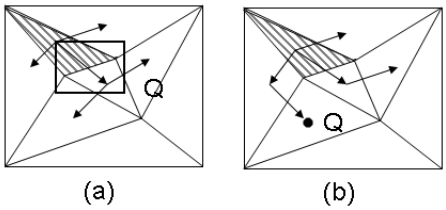


**Figure 7: (a) A 2D example of Directed Local Search. (b) Directed Local Search for a point query.**

## 3.3 Z-Order based techniques

Z-order based techniques overlay a rectilinear grid on the indexed domain. Each object is approximated by a collection of grid cells and a linear cell ordering is computed using the Z-order [26, 27].

The cells intersecting the query range (or the cell containing a query point) is identified by comparing the query's Z-order value range with those of the cells, using a B-Tree. Query performance depends on the grid resolution. A very fine grid will lead to a large number of indexed cells, but each cell will overlap with only a few elements. Higher resolution reduces the number of cells at the cost of losing precision and performance.

Investigating the tradeoffs involved in picking the right resolution and thus minimizing the impact of replication is an interesting exercise. However, regardless of the optimal resolution, the z-order approach will suffer from the same inherent problems of replication, described in Section 3.2.

Table 1 demonstrates the amount of required replication for varying grid resolutions, for the uniform, dense region of the dataset in Section 3.1. Figure 1 shows the average number of tetrahedra overlapping a cell. For the high resolution grid, which consists of $128^3$ cells, the dataset requires $128^3 \times 10 = 20971520$ records, which implies a $20\times$ increase in the dataset size. Lower resolutions reduce the storage overhead, at the expense of search time (100 elements must be searched for the $16 \times 16 \times 16$ decomposition).

# 4. DIRECTED LOCAL SEARCH

In this section we present Directed Local Search (DLS). We first present the basic DLS algorithm for range and point queries and then describe in detail the techniques that enable DLS, namely the proximity search algorithm and the compressed representation of mesh connectivity information.

## 4.1 Algorithm Overview

Directed Local Search processes range and point queries by utilizing the mesh connectivity. Figure 7 (a) shows an example range query on a 2D triangular mesh. If we know a single initial element that is part of the answer (highlighted) we compute the query result by searching first all of the initial element's neighbors and incrementally expanding the search in a breadth-first search (BFS) fashion until we find no additional elements within the range. Figure 7 (b) shows the same principle applied to a point query: Starting at an initial element, we perform the same expansion until we reach the target element.

```
1. Identify a suitable starting element E
2. enqueue (BFS_Queue, E); mark E as visited
3. While (BFS_Queue not empty)
   3.1 element e = dequeue(BFS_Queue)
   3.2 if e intersects Q
     3.2.1 add e to the query result
   3.3 for each ni in neighbors(e)
     3.3.1 if face fi intersects Q
               and ni not visited
     3.3.2  enqueue (BFS_Queue, ni)
     3.3.3  mark ni as visited
```

**Figure 8: The Directed Local Search algorithm**

Figure 8 details the DLS algorithm for a range query $Q$. Step 1 identifies a starting element that intersects the range query, using the *proximity search* algorithm described in the next section. The next steps describe the breadth-first search (BFS), that stops when no further elements intersecting the range can be found. The predicate in step 3.3.1 examines if the face $f_i$ of the current element $e$ intersects the query range. If it does not, it is not necessary to visit neighbor $n_i$.

The primary advantage of DLS over traditional spatial indexing is that the breadth-first search is independent of the mesh geometry. Using the mesh connectivity avoids the performance problems generated by overlapping MBRs. Furthermore, DLS does not approximate tetrahedra by simpler shapes and therefore directly computes the query results instead of first forming approximate answers and then post-processing them.

Implementing DLS requires solving the following subproblems. First, we need a way to determine a suitable starting element (step 1 of Figure 8) that intersects the query range. We call this starting element selection "proximity search" and describe it in Section 4.2. For point queries, the proximity search described in the next section directly finds the containing element and thus the BFS search of Figure 8 is not used. In addition, we need to efficiently represent the "neighbor" relationships between elements, so that BFS can quickly access them. In Section 4.3 we show how to improve over the adjacency-list representation of connectivity information.

Finally, note that DLS is guaranteed to succeed if the dataset is convex, not containing any holes or concavities. This is the case for many models used in practice (ground, materials models). In practice, our techniques work also for small holes or concavities because the BFS can "work around" them as long as they are not too big. A more general treatment involves cataloguing all the exterior mesh surfaces and "jumping" from one surface face to another as long as they are contained in the query region. Such a solution is part of our ongoing work.

## 4.2 Proximity Search

In this section we present algorithms for selecting an ini-

| Query | Point | Range 1% | Range 5% | Range 10% |
|---|---|---|---|---|
| "hit" %' | 52% | 93.3% | 94.7% | 97.7% |

**Table 2: BFS "hit rates" for various query types**

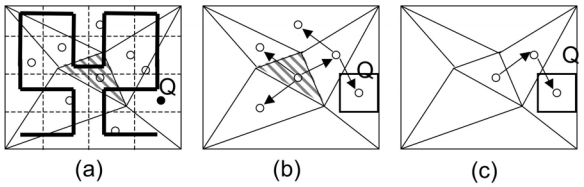


**Figure 9: (a) Using the Hilbert order to select a starting element for point Q. (b) Hilbert_BFS example. (c) Hilbert_Direct example.**

tial "seed" element for DLS. We use the Hilbert curve to index the tetrahedral elements, after representing each element by its center point. For a range query, we find the element whose center has the closest Hilbert value to the center of the query range. For a point query, we identify the element closest to the query point in Hilbert space (Figure 9 (a)). We work similarly for range queries, looking for an element close to the center of the query range. The distance preserving properties of the Hilbert curve [25] imply that the obtained element will be close to the query region and thus a suitable starting point.

The Hilbert ordering translates the proximity search, a geometric operation, into a numerical one. Thus we are able to use linear access methods (the B-Tree) that are fast and predictable, avoiding the complexity and cost of accessing a spatial access method.

To our knowledge this is the first time that the Hilbert curve is directly used for spatial indexing. (Multiple Hilbert curves have been proposed for nearest neighbor queries [22]). The problem is that there exist no guarantees about the distance between the returned element and the query point. The novelty of our approach is that using the connectivity information, we can correct the problem by reaching a suitable starting element even if the Hilbert index returns an element that is far away.

Figure 9 (b) outlines our basic proximity search algorithm, Hilbert_BFS. Hilbert_BFS selects an initial element using the Hilbert value index and if it doesn't overlap the query region, it performs a breadth-first expansion until another suitable element is found. Regardless of the initial element the algorithm will eventually return an overlapping element, as in the worst case the entire dataset will be scanned. The Hilbert ordering of the elements benefits BFS by increasing spatial locality and minimizing page accesses.

Figure 9 (c) presents Hilbert_Direct, an improvement over the basic Hilbert_BFS algorithm. Instead of "expanding" the search towards all directions, Hilbert_Direct follows a path of elements towards the center of the query region. The next neighbor in the path is determined by "drawing" a line connecting the center of the current triangle to the center of the query region and crossing the face intersected by the line. Hilbert_Direct resembles a depth-first search, since it expands only the one neighbor that lies in the direction of the query region. It is still useful to remember the other neighbors as well, since following alternative paths makes the algorithm robust to small concavities and "holes".

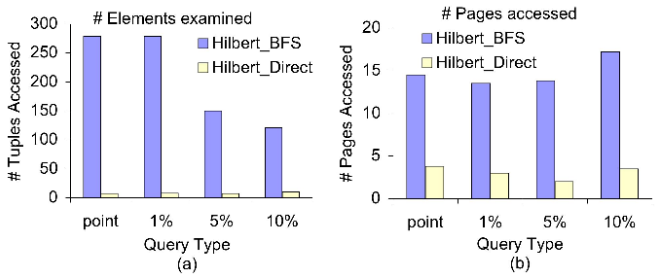


**Figure 10: (a) Elements examined by the BES variants. (b) Corresponding page accesses**

We now present a characterization of the above algorithms in terms of their effectiveness in determining a suitable starting element, using our *gear* dataset discussed in Section 6. We answer the following questions:

1. How often does the Hilbert index provide a suitable starting element immediately? For range queries, we measure the percentage of queries that immediately find an intersected element. For point queries, we measure the percentage of queries where the returned element contains the query point.
2. How many elements does each of the above techniques examine before a suitable element is returned?
3. How many *page accesses* does each of the above search techniques require?

Table 2 shows how effective the Hilbert index would be if it was used by itself. The Hilbert index immediately returned the correct element for half of the point queries. Also, more than 90% of the queries can be answered by the Hilbert index directly. Thus Hilbert indexing by itself is highly efficient, returning immediately suitable results most of the time. Similar results were obtained for all the datasets described in our experimental section.

Figures 10 (a)-(b) characterize the performance of Hilbert_BFS and Hilbert_Direct, by measuring the tuples and pages accessed until we reach a suitable starting element, considering only the queries that *were not answered immediately by the Hilbert index*. We consider point queries and range queries whose size is 1, 5 and 10 percent of the dataset size. Hilbert_BFS accesses a much higher number of tuples than Hilbert_Direct, because it searches towards all directions. Since Hilbert_Direct offers the best performance, we use it in our implementation and experimental evaluation.

Hilbert_Direct uses the same geometric principles as the "triangulation walking" studies [6] by the computational geometry community, where the main focus is the theoretical analysis of point query performance. A complete theoretical analysis is a separate area of study, where the additional assumption of uniformly distributed mesh nodes is necessary for tractability [7] (such a theoretical analysis is beyond the scope of this paper). When running point queries, at most 128 elements were processed by a single query before finding a suitable starting point, incurring 9 page accesses. In our experiments, only one in a thousand queries exhibits this worst-case scenario, hence the excellent average performance shown in Figure 10.

## 4.3 Representing Element Adjacency

In order to implement the algorithms of the previous section we need to be able to retrieve the neighbors for each mesh element. A simple way to obtain this connectivity information is through the mesh generation process itself: Mesh generators like *Pyramid* [30] can provide the neighbors of each tetrahedral element as part of the output. If the connectivity information is not accessible, there are simple techniques to compute it, by using a hash table to match the elements with the same face. There is even a way to compute element connectivity using a standard commercial DBMS, as outlined in [15] [2]. The development of optimized ways to extract connectivity information from large meshes is part of our ongoing work.

The connectivity information comprises, for each element, the location on the disk of its 4 (at most) neighbors. A disk pointer is a (*page_no*,*offset*) tuple ID. The simplest way to store the pointers is to extend the *Elements* table with 4 additional columns. However, it is desirable to develop a more efficient representation technique for the connectivity information in order to improve the storage requirements and I/O performance of our solution.

We propose a compressed representation based on the clustering properties of the Hilbert curve. By computing the Hilbert ordering for the elements, besides facilitating the efficient proximity search of Section 4.2, we *re-label* the elements so that the spatially close elements receive IDs that tend to be numerically close. The implication is that the neighbors of an element are also likely to receive similar IDs. We take advantage of this labeling property by actually storing the *differences* between the IDs of an element and its neighbors. The motivation is that in the common case the difference will be much smaller than the IDs themselves and thus, with an appropriate encoding scheme, it will require fewer bits. Figure 11 shows an example. The neighbors of element "1000" received similar IDs. The ID differences are orders of magnitude smaller and require fewer bits to represent.

We now describe our compression scheme in more detail. Given an element with ID $E$ and 4 neighbors with IDs $E_1$, $E_2$, $E_3$, $E_4$, we encode them by storing the values $code(E - E_1)$, $code(E - E_2)$, $code(E - E_3)$, $code(E - E_4)$ in a variable length field. For the compression to be efficient, the integer encoding function $code(.)$ must represent small values with fewer bits compared to larger values.

We use *snip* codes [4], shown to provide both high storage efficiency and performance for general purpose graph compression. The snip encoding of an integer is the actual binary representation of the integer, in a linked list format. Each 2-bit "snip" contains 1 binary digit from the number's representation and one "continue" bit that is set to zero only for the last snip. We use one additional snip for signs, as ID differences could be negative and we prefix the code with 2 bits denoting the neighbor count.

In practice, instead of logical IDs, we need to store the differences of page numbers (encoding the offset within the page is easy, as it is typically small). Compressing page differences is not straightforward. We need to know the page numbers of neighboring elements in advance, but this is impossible as the page number of an element depends in turn on the compression of all the previous elements!

[2]By generalizing their surface extraction algorithm.

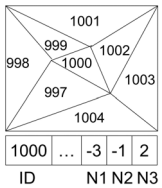


**Figure 11: 2D example of adjacency compression. Logical ID differences require fewer bits.**

| Dataset | Rows | Bits/Record | Fragmentation |
|---|---|---|---|
| gear | 8.8M | 14.5 | 3% |
| circle | 10M | 14.3 | 3.1% |
| cube | 5.4M | 14.8 | 3.4% |
| heart | 510K | 12.2 | 2.9% |
| quake | 14M | 14.2 | 3.3% |

**Table 3: Compression results for our real datasets.**

For simplicity, instead of developing complex multi-pass mapping algorithms we use a constant page capacity $P$ for the entire dataset. We set $P$ to be equal to the minimum page capacity when we use logical ID encoding. Since the page number differences are smaller than the ID differences, this approach might generate some free space in the page. We can minimize the free space by increasing $P$ an recomputing the differences, up to the point where we do not cause page overflows.

Table 3 quantifies the compression achievable for our experimental datasets and the amount of internal fragmentation incurred and characterizes the storage overhead of our technique. The connectivity information requires fewer than 2 bytes per record even for large datasets, in contrast to the 24 bytes of the adjacency-list implementation. The small overhead (increased space utilization) translates to improved I/O performance for range queries.

## 4.4 DLS Generalization

DLS can form the basis for a more general indexing tool that can support other types of mesh shapes besides the tetrahedra, such as bricks, prisms, or pyramids with more faces (like pentahedra). DLS is extensible because for all of the above shapes we can exploit the foundations of our technique:

1. There exists a Hilbert encoding, that allows for proximity search and clustering.
2. We can easily evaluate point containment or range intersection predicates.
3. We can exploit element connectivity.

Extending DLS to handle *finite volume* meshes, such as those used in Computational Fluid Dynamics (CFD) is not as straightforward, because such meshes are often represented in a face-oriented rather than element-oriented fashion. The absence of explicit elements means that are no constraints in the volumes enclosed by faces, allowing nonconvexities and multiple elements sharing a face. Furthermore, there are numerical tolerance issues in determining whether a face contains a query point. A solution to this problem requires the development of new indexing techniques and is also part of our ongoing work.

# 5. GRAPH-BASED CLUSTERING FOR TETRAHEDRAL MESH DATA

Our techniques use the Hilbert curve to cluster mesh elements on the disk, minimizing the *Elements* page accesses. In this section we show that, while Hilbert clustering offers very good performance for rectangular, box-shaped range queries, it is sub-optimal for queries on arbitrarily shaped regions (like surfaces) that are frequent on scientific applications.

To improve the clustering for these cases, we introduce the idea of graph-based clustering. Rather than relying on element center coordinates, graph based techniques try to place an element on the same page with its neighboring elements, as frequently as possible.

Our approach goes beyond space-filling curve clustering, so far the only general-purpose layout technique for spatial data, by allowing efficient retrieval of arbitrary, application-specific regions *without* sacrificing the overall spatial locality of the layout (and thus without requiring multiple copies or orderings of the same data and without affecting DLS indexing).

## 5.1 Graph Partitioning and I/O

We use the *dual graph* representation of a mesh, mapping each mesh element to a graph vertex and each pair of neighboring elements to an edge. Using the dual graph, we restate the abstract problem of "preserving element spatial locality" as a graph partitioning problem: We partition the dual graph vertices into page-size chunks so that we minimize the number of edges crossing page boundaries. This formulation implies spatial locality because, intuitively, nodes connected by an edge are likely to be retrieved together by a range query. .

Graph partitioning is a hard problem [9] but there exist many practical heuristics. We use METIS [19], a multi-level graph partitioning heuristic, shown to offer the best known results [4]. The dual representation of a mesh is given by the mesh generation process and thus we can use METIS directly. In the case of very large models, we can first coarsely partition them into memory-sized chunks using the Hilbert clustering and use METIS on each individual partition. Alternatively, we could modify METIS to produce memory-sized first level partitions.

## 5.2 Feature-Based Clustering

Scientific datasets commonly have distinct *features*, connected regions repeatedly queried by the application. An example is the ground surface in earthquake simulations, as we are interested in the damage inflicted on the buildings. Features are usually known in advance and are heavily queried because we need to access the relevant data *for every simulated time step*.

Hilbert curve clustering is suboptimal for querying features, because it is optimized for rectangular, box-like queries. It is well known that its quality deteriorates with increasing query region hyper-surface [25]. We use graph-based clustering to improve the retrieval performance for feature queries. Our approach is based on *explicitly* specifying frequently co-accessed elements and on using this information to guide data layout. The dual graph of the mesh dataset is ideal for this purpose, as co-access information can be encoded into the edge weight.

As the example in Figure 12 shows, we overlay the dual

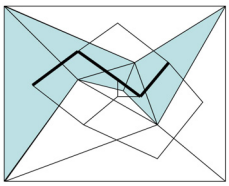

**Figure 12: A 2D feature example (highlighted)**

graph on the feature region and strengthen the weights for the edges within the region. Our experimental results suggest that it is sufficient to increase the edge weights so that the total weight associated with the region is is comparable to the total original weight of the entire graph. If the weight increase is too small, it will not affect the overall quality of the solution. Due to the increased edge weights, graph partitioning will pack the feature's elements in disk pages. sacrificing some of the edges not fully contained in the feature. The remaining "regular" edges help by still maintaining the overall spatial locality in the dataset.

Note that feature based clustering is not related to indexing: The feature specification and the assignment of elements to features is application dependent. Rather than identifying the feature elements, feature-based clustering reduces the I/O cost of *retrieving* them after they have been identified by the application. Also, although we are motivated by datasets queried repeatedly (per time-step), in this paper we only consider spatial queries (not involving multiple time-steps).

# 6. EXPERIMENTAL SETUP

In this section we describe the techniques, datasets and methodology used to experimentally evaluate DLS.

## 6.1 Implementation

We implemented DLS on top of PostgreSQL (version 7.4.5). The data is stored in the *Elements* and *Nodes* tables as shown in Section 2. We store element center coordinates using the *cube* datatype included with the PostgreSQL distribution and use the center for sorting and building the B-Tree for the proximity search, based on the Hilbert ordering. To incorporate the Hilbert order, we replaced the PostgreSQL comparison routines with code that compares Hilbert values *directly* from IEEE double precision coordinates, without actually computing them. The direct comparison routines allow us to use the highest possible Hilbert curve resolution, equivalent to 192-bit Hilbert values. The additional storage required for the connectivity information is shown in Table 3 of Section 4.3. The DLS routines were implemented as new join operators, based on the PostgreSQL nested loop join.

We compare DLS to a Hilbert R-Tree implementation also built on top of PostgreSQL. We utilized the GiST access method, which we modified to allow for the Hilbert R-Tree bulk loading method. The Hilbert R-Tree is optimized so that it stops the search once the containing element for a point query is found, eliminating unnecessary page accesses.

Our experiments run on a 2-way P4 (3.6 GHz, 2MB L2) Xeon machine with 4GBs of memory and 2 320 SCSI-2 hard drives, running Linux 2.6.

| Name | Elements | Nodes | Size (GB) | R-Tree levels | B-Tree levels |
|---|---|---|---|---|---|
| gear | 8.8M | 1.3M | 1.3 | 4 | 4 |
| circle | 10M | 1.5M | 1.4 | 4 | 4 |
| cube | 5.4M | 0.9K | 0.6 | 4 | 4 |
| heart | 570K | 110K | 0.1 | 3 | 3 |
| quake | 14M | 2.5M | 2 | 4 | 4 |

**Table 4: Datasets used in our experiments.**

| | |
|---|---|
| rtree | HRTree pages accesses. |
| btree | B-Tree pages accesses (DLS). |
| elements | The *Elements* page accesses. |
| nodes | Accesses to *Nodes* and its index. |

**Table 5: PostgreSQL page access categories.**

## 6.2 Datasets and Queries

We experiment with the real 3D mesh datasets shown in Table 4. The *gear*, *disk* and *cube* meshes are used in crack propagation whereas the *quake* meshes are used in earthquake simulations. Finally, the *heart* dataset is a model of a human heart developed for use in biomedical applications[3].

Our query workloads consist of uniformly distributed point and range queries. We vary the sizes of the range queries, so that the range size is equal to 1%, 5% and 10% of the dataset size.

## 6.3 Performance Metrics

We report page accesses and query running times for the point and range queries described in the previous sections. For page access counts, we report the number of *distinct* database pages accessed per query, broken down into the subcategories of Table 5. We measure running times on a "cold" system, where no data or index pages are cached in main memory at the beginning of each query. We use cold measurements because they better capture the impact of I/O on query performance. We break the running times into the components shown in Table 6.

Performance improvements in terms of *speedups* of DLS over the Hilbert R-Tree (HRTree) are computed by:

$1 - Page\ accesses\ (DLS)/Page\ accesses\ (HRTree)$ and
$1 - Running\ Time(DLS)/Running\ Time(HRTree)$.

Constructing the Hilbert index is essentially a standard sorting operation that uses the comparison routine of Section 6.1. It is handled by the DBMS and we therefore do not report its performance. Besides, the Hilbert R-Tree which we compare against has exactly the same performance. The connectivity information is computed and stored during the mesh creation time and involves an additional pass over the *Elements* table for computing the new element IDs after sorting the elements according to the Hilbert order.

# 7. EXPERIMENTAL RESULTS

In this section we experiment with DLS and compare its performance against the Hilbert-packed Tree (HRTree).

[3]Made available by the Computational Visualization Center at the University of Texas, Austin (http://ccvweb.csres.utexas.edu/cvc/).

| | |
|---|---|
| rtree | The time for an R-Tree lookup. |
| btree | The time for a B-Tree lookup. |
| nodes | Time for a lookup on the *Nodes* table. |
| elements | Time for DLS proximity search and BFS. |
| others | Containment predicate, direction computation. |

**Table 6: Running time breakdowns.**

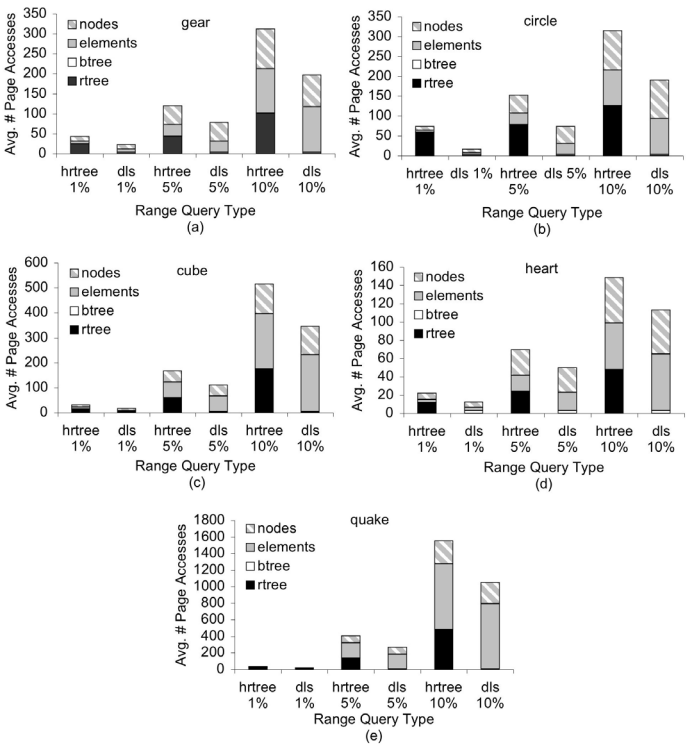


**Figure 13: Page accesses for range queries of varying sizes on our 5 datasets for the Hilbert R-Tree and DLS.**

## 7.1 Range Query Performance

Figures 13 (a)-(e) show the average number of page accesses for range intersection queries of varying sizes, on 5 datasets. For each pair of bars, the first corresponds to the HRTree (labeled rtree) and the second to DLS (dls). As Figure 13 demonstrates, the R-Tree access is the largest component in all the cases (up to 74% for *circle* 1%) except for the "large" 10% queries where it is comparable to the *Elements* table accesses. For the 10% queries, the R-Tree is responsible for 32% (*heart*) to 39% (*circle*) of the accesses.

DLS eliminates the R-Tree overhead by combining an efficient B-Tree lookup and a localized proximity search instead of a costly R-Tree traversal operation. The B-Tree lookup requires 4 page accesses for all datasets (except for *heart* with only 3 levels). The proximity search is highly efficient, requiring 0.5-2 additional page accesses on average across all datasets. The overall effect is a reduction of up to 96 times (*quake* 10%) in the number of page accesses required for indexing.

The improvement is larger for our more complex datasets (*gear*, *circle*, *quake*) as opposed to the more uniform ones (*cube*, *heart*), highlighting the robustness of DLS with respect to the geometric complexity of the dataset (the same

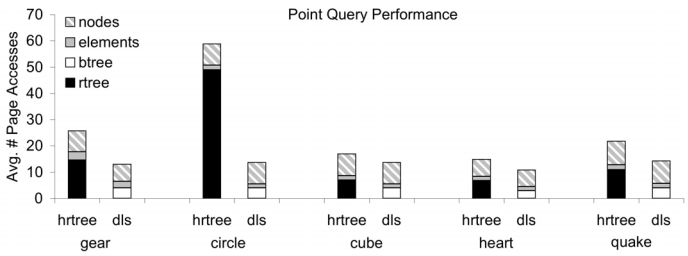


**Figure 14: Page Accesses per point query for the Hilbert R-Tree and DLS on our 5 datasets.**

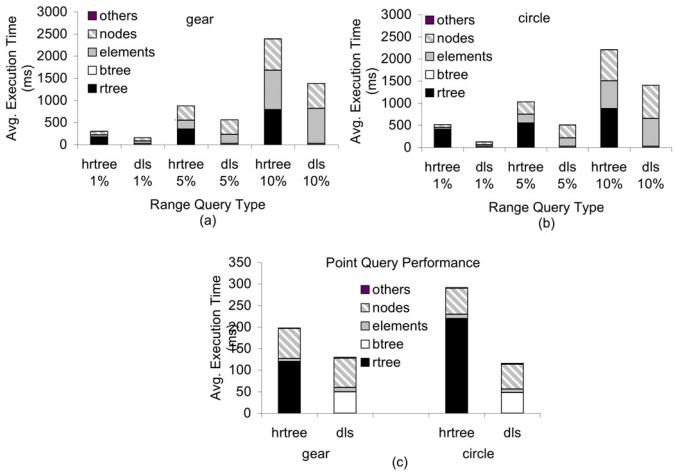


**Figure 15: (a)-(b) query execution times for range queries on *gear* and *circle*. (c) Query execution times for point queries.**

trend appears in the results for point queries, Section 7.2). The *cube* and *heart* meshes contain more regular elements (in terms of size/geometry), characteristics that help the performance of the HRTree. The *gear*, *circle* and *quake* datasets are representative of the real meshes used in applications: In practice only certain regions of the domain are "refined" (modeled with large numbers of tiny elements), which leads to significant irregularities in the mesh structure.

The number of *Elements* pages accessed by DLS is comparable to that of the HRTree, even for the 10% queries. Due to the effectiveness of our compression technique, the additional connectivity information does not deteriorate I/O performance. For all the "small" ranges (1%) DLS accesses 1-3 fewer pages than the HRTree, corresponding to a reduction of up to 48%. This happens because the R-Tree accesses leaf pages whose MBR intersects the query range, but do not contain a result. The impact of this imprecision decreases for larger ranges, as it is more likely that a leaf page will actually contain an element intersecting the query range and is anyway needed by the query.

The overall performance improvement offered by DLS ranges from 28% to up to a significant factor of 4 (*circle*, 1%). The *circle* dataset benefits from DLS the most, with improvements of 36% up to a factor of 4.

Figure 15 (a), (b) shows the query execution times for the same workloads on the *gear* and *circle* datasets. The query

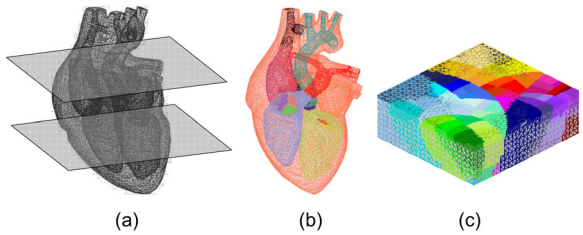


**Figure 16: (a) Heart model with cross-sections. (b) Heart model with boundaries. (c) Partitioned ground mesh**

execution times confirm the trend in the page access count results. DLS can improve overall query execution performance by up to a factor of 4 for *circle* 10%.

## 7.2 Point Query Performance

Figure 14 shows the average number of page accesses for point queries. The results are similar to the range query results. R-Tree accesses correspond to 41% (*cube*) to 83% (*circle*) of the total page accesses. As in Section 7.1, *circle* and *gear*, the most irregular datasets in this study, have the highest number of R-Tree accesses.

DLS replaces the expensive R-Tree lookups with B-Tree page accesses that need only 4 page accesses (3 for the smaller *heart*). The elimination of the R-Tree leads to overall improvements ranging from 19% up to a factor of 4.

The number of *Elements* pages accessed by the R-Tree and DLS methods is very similar, 1.67-2.5 for the HRTree and 1.4-1.64 for DLS. DLS accesses slightly fewer pages on average, as the R-Tree might have to access more elements simply because their MBRs intersect the query point, without actually being part of the solution. The Hilbert clustering of *Elements* however helps in keeping those elements on the same page.

Figure 15 (c) shows the query response times for point queries on two datasets, *gear* and *circle*. The execution times confirm our page count measurements and demonstrate that the reduction of R-Tree page accesses by DLS can lead to significant savings in the overall query response time. Similar to our page access results, *gear* and *circle* show the largest running time improvements, 34% up to a factor of 2.5.

## 7.3 Feature Clustering: Heart Model

In this section we show that feature-based clustering provides better I/O performance compared to Hilbert curve clustering for feature queries. We use two feature examples on the *heart* dataset.

Figure 16 (a) shows our first experiment, a situation where several cross-sections of the heart model have been identified in advance and are used for querying the model.

We use 10 such cross-sections, each represented by a "thin" range query, with height equal to 1% of the dataset height, randomly spread within the model. The elements intersected by each cross-section are known in advance and our goal is to reduce the number of page accesses required to retrieve them.

Figure 17 (a) compares the performance of Hilbert-order clustering to that obtained by using the ideas in Section 5.2 and the METIS partitioning tool. Feature-based clustering

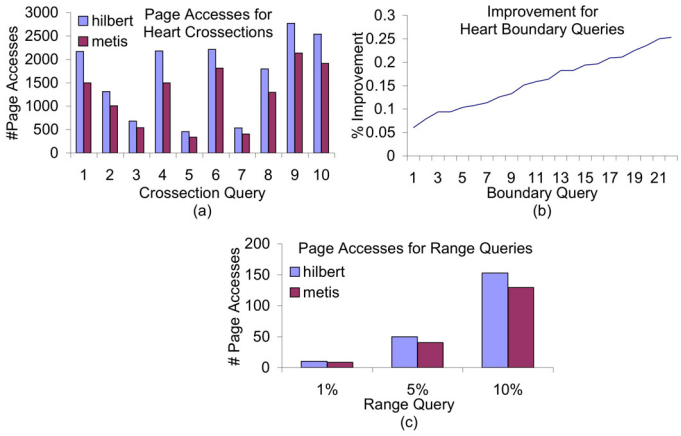


**Figure 17: (a) Page accesses for retrieving the 10 cross-sections. (b) % Improvement for retrieving boundaries. (c) Average accesses for range queries.**

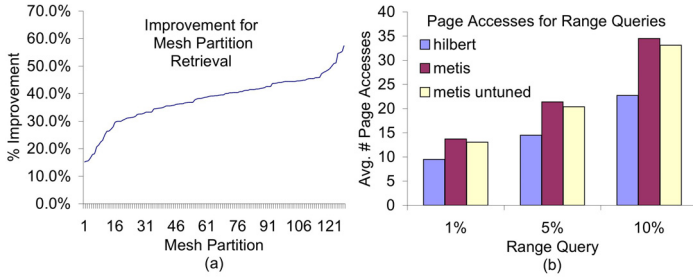


**Figure 18: (a) % Reduction in the number of page accesses per partition for the earthquake dataset. (b) Average page accesses for range queries**

reduces the number of pages accessed per cross-section by 18%-31%, for an average improvement of 25%. More importantly, as Figure 17 (c) demonstrates, this performance improvement comes at no cost for the average spatial range query performance. In fact, the layout obtained through graph partitioning improves range queries by 15%-19%.

For this particular scenario the Hilbert curve clustering could be modified to favor accesses along horizontal cross-sections, by appropriately "stretching" the Hilbert cells. This approach however does not work for arbitrarily shaped features. As the next example illustrates, feature-based clustering is a natural match for such datasets.

Figure 16(b) shows the heart model augmented with additional surfaces corresponding to various heart components (like the pulmonary valve or the aorta). This information is provided by the model's constructor by assigning each *node* to either one of 22 different *boundaries* or to the model's interior. Using the node information we identify the elements adjacent to each boundary and treat each boundary as a separate feature.

Figure 17 (b) compares the page accesses per surface for the Hilbert and feature-based clustering. The improvements obtained by our layout are 6% to 25% for an average of 16%. Again, like in the previous example. there is no impact on the performance of the random range query (Not shown).

## 7.4 Feature Clustering: Earthquake model

Figure 16 (c) shows a ground model mesh used in earthquake simulations. The mesh is partitioned into 128 parts for parallel simulation on 128 computing nodes. Improving the performance of retrieving the individual mesh components is useful for applications that move the same data between the storage subsystem and the computation nodes multiple times, like for example large-scale I/O-bound visualization systems [35] that read the same partitions for every time-step and distribute them to rendering processors working in parallel.

Figure 18 (a) shows the reduction in the number of page accesses obtained by feature-based clustering over the Hilbert clustering for the 128 partitions. The improvements range from 15.2% to 57.5% with an average improvement of 37.9%.

Figure 18 (b) shows the impact of the graph-based clustering on the performance of random range queries of varying sizes. Contrary to the previous examples, feature based clustering in this case hurts random query performance. This happens because Hilbert clustering has much better performance compared to graph based techniques for this particular dataset, even when we use graph partitioning on the initial dual graph *without* changing any edge weights (as the bar labeled *metis_untuned* of Figure 18 (b) demonstrates).

We believe that this happens because the mesh model we used is small (150K elements) and regularly structured. It is derived by triangulating an oct-tree mesh, thus the elements fit nicely into cubical regions and the Hilbert curve does a better job at clustering them. This example, besides highlighting the potential for improving Hilbert-based clustering, motivates further research on the general properties of graph-based partitioning, specifically on how it relates to different dataset geometries.

# 8. CONCLUSION

In this paper we examine database support for efficient query execution on large tetrahedral mesh datasets. We present Directed Local Search (DLS), a query processing technique for spatial queries that takes advantage of the mesh connectivity and its efficiency is independent of the complexity of the mesh geometry. We show that DLS can be easily implemented in a database system without requiring the development of new access methods. We also propose a new graph-based technique for clustering mesh elements to disk pages and demonstrate that it has better performance than traditional clustering techniques using space-filling curves when retrieving regions of arbitrary shapes.

# 9. ACKNOWLEDGEMENTS

This work is sponsored by the National Science Foundation under Grants CNS-0509004, IIS-0429334, SEI+II-0431008, 0429334, and 0329549, by a subcontract from the Southern California Earthquake Center's CME Project as part of NSF ITR EAR-01-22464, by support from the Intel Corporation, Carnegie Mellon CyLab and CyLab Korea, by Microsoft Research and the DARPA SIPS program, by a Sloan fellowship, an IBM faculty partnership award and a NASA AISR fund.

The authors would like to thank Jacobo Bielak and Omar Ghattas for their long term collaboration on the Carnegie Mellon Quake project, Greg Ganger and the entire Parallel Data Lab staff for their help in building our storage and

compute server infrastructure, and the anonymous reviewers for their insightful comments.

## 10. REFERENCES


[1] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terascale computers. In *Proceedings of Supercomputing 2003*.

[2] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-Tree: a practically efficient and worst-case optimal R-Tree. In *Proceedings of SIGMOD'04*.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

[4] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *SODA '03*, pages 679–688, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[5] J. V. den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of VLDB '97*, pages 406–415, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[6] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. In *Proceedings of SCG '01*, pages 106–114, New York, NY, USA, 2001. ACM Press.

[7] L. Devroye, C. Lemaire, and J.-M. Moreau. Expected time analysis for delaunay point location. *Comput. Geom. Theory Appl.*, 29(2):61–89, 2004.

[8] V. Gaede and O. Guenther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[10] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. S.Szalay, D. DeWitt, and G. Heber. Data managemenet in the coming decade. Technical Report MSR-TR-2005-10, Microsoft Research, 2004.

[11] O. Guenther. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proceedings of ICDE'89*, pages 598–605, Washington, DC, USA, 1989.

[12] A. Guttman. R-Trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57. ACM Press, 1984.

[13] M. Hadjieleftheriou. *www.cs.ucr.edu/ marioh/spatialindex*.

[14] G. Heber and J. Gray. Supporting finite element analysis with a relational database backend part i: There is life beyond files. Technical Report MSR-TR-2005-49, Microsoft Research, 2005.

[15] G. Heber and J. Gray. Supporting finite element analysis with a relational database backend part ii: Database design and access. Technical Report MSR-TR-2006-21, Microsoft Research, 2006.

[16] H. V. Jagadish. Spatial search with polyhedra. In *Proceedings of ICDE'90*, pages 311–319, 1990.

[17] J.Gray, D.Slutz, A.Szalay, A.Thakar, J.vandenBerg, P.Kunszt, and C.Stoughton. Data mining the SDSS skyserver database. Technical Report MSR-TR-2002-01, Microsoft Research, 2002.

[18] I. Kamel and C. Faloutsos. On packing R-Trees. In *Proceedings of CIKM'93*.

[19] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[20] D. Komatitsch, S. Tsuboi, C. Li, and J. Tromp. A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the earth simulator. In *Proceedings of the Supercomputing Conference*, 2003.

[21] S. Leutenegger, J. Edgington, and M. Lopez. STR: A simple and efficient algorithm for R-Tree packing. In *Proceedings of ICDE 1997*.

[22] S. Liao, M. A. Lopez, and S. T. Leutenegger. High dimensional similarity search with space filling curves. In *Proceedings of ICDE 2001*.

[23] J. C. Lopez, D. R. O'Hallaron, and T. Tu. Big wins with small application-aware caches. In *Proceedings of Supercomputing '04*, page 20, Washington, DC, USA, 2004.

[24] K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of PRS '95*, pages 23–30, 1995.

[25] B. Moon, H. v. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE TKDE*, 13(1):124–141, 2001.

[26] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of SIGMOD '89*, pages 295–305, 1989.

[27] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS '84*, pages 181–190, New York, NY, USA, 1984. ACM Press.

[28] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-Trees. In *SIGMOD '85*, pages 17–31, New York, NY, USA, 1985. ACM Press.

[29] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.

[30] J. R. Shewchuk. Tetrahedral mesh generation by delaunay refinement. In *Symposium on Computational Geometry*, pages 86–95, 1998.

[31] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization'02*, 2002.

[32] Office of Science Data Management Workshops. The office of science data-management challenge. Technical report, Department of Energy, 2005.

[33] T. Tu and D. R. O'Hallaron. A computational database system for generatinn unstructured hexahedral meshes with billions of elements. In *Proceedings of Supercomputing '04*.

[34] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transctions VCG'96*, 3(4):370–380, 1997.

[35] H. Yu, K.-L. Ma, and J. Welling. A parallel visualization pipeline for terascale earthquake simulations. In *Proceedings of Supercomputing '04*.