The Hilbert PDC-tree: A High-Velocity Structure for Many-Dimensional Data

David Robillard
School of Computer Science
Carleton University
Ottawa, Canada
d@drobilla.net

Frank Dehne
School of Computer Science
Carleton University
Ottawa, Canada
frank@dehne.net

Andrew Rau-Chaplin
Faculty of Computer Science
Dalhousie University
Halifax, Canada
arc@cs.dal.ca

Neil Burke
Faculty of Computer Science
Dalhousie University
Halifax, Canada
Neil.Burke@dal.ca

ABSTRACT

Fast aggregation of data with many dimensions is a key component of many applications. The R-tree is the traditional data structure for indexing multi-dimensional data, but even the best R-tree variants suffer from performance degradation as the number of dimensions increases. The DC-tree addressed this issue by replacing Minimum Bounding Rectangle (MBR) keys with Minimum Describing Subsets (MDSs), which are less susceptible to overlap. This technique dramatically improves query performance with many dimensions, but at the cost of reduced insertion performance. Like most R-tree variants, this insertion overhead comes from expensive geometric comparisons while selecting the best child for insertion, or splitting over-full nodes. DC-trees, including the parallel PDC-tree, suffer even more from this overhead since MDSs are typically much more expensive to compare and manipulate than MBRs. This paper introduces the Hilbert PDC-tree, a parallel index structure for many-dimensional data that supports high-velocity data ingestion. This is achieved by avoiding geometric comparisons during insertion by instead inserting records based on the Hilbert index of their keys. This approach is similar to that of the Hilbert R-tree, but with special considerations for efficiently supporting many hierarchical dimensions. Additionally, a new node splitting algorithm significantly reduces overlap and improves query performance. Experiments show that the Hilbert PDC-tree scales well to a high number of dimensions, while supporting a much higher rate of ingestion and better query performance than the PDC-tree.

1. INTRODUCTION

Many applications, from traditional OLAP systems to emerging frameworks designed for more ad-hoc data, rely on the ability to quickly aggregate "slices" of multi-dimensional data sets with dimension hierarchies. Recent times have seen increased interest in the ability to do so in real-time, to facilitate live analysis of high-velocity data streams.

Dynamic systems with a need to index multi-dimensional data sets typically use some variant of the classical R-tree. R-trees are an effective solution for indexing a small number of dimensions, but their query performance rapidly degrades when the number of dimensions becomes large. This effect is primarily due to overlap between the bounding boxes for tree nodes, which MBRs are particularly susceptible to with many dimensions.

The DC-tree [6] addressed this limitation of R-trees by replacing MBRs with MDSs, and achieves much better query performance as the number of dimensions gets very large. However, this ability comes at the cost of increased overhead, particularly during data ingestion since the cost of insertion is much greater than that of an R-tree. The PDC-tree [5] extended the DC-tree with support for multi-threading, providing speedup on multi-core architectures, but with the same underlying costly insertion algorithm.

This paper introduces the Hilbert PDC-tree, which is designed to preserve the benefits of the PDC-tree while supporting much higher velocity data ingestion. The use of the Hilbert curve, rather than geometric comparisons, allows the insert position for a new element to be found much more quickly. MDS keys in conjunction with the locality-preserving Hilbert mapping provide good query performance with many dimensions. Additionally, a new split algorithm minimizes overlap within the constraints imposed by the Hilbert ordering, to facilitate efficient query execution.

The Hilbert PDC-tree supports the use of hierarchical dimensions, as in OLAP systems, or other data sets with hierarchical ontologies. The implementation described in this paper was designed to support indexing hierarchical point data, that is, keys with a single value in each dimension. However, the fundamental ideas are more general, and can easily be adapted for other scenarios, such as non-hierarchical dimensions, or indexing elements whose keys have volume (i.e. are not points).

The relevant related work and necessary background concepts are described in Section 2. The novel aspects of the Hilbert PDC-tree are described in Section 3. Experiments in Section 4 demonstrate performance in detail, and confirm that the Hilbert PDC-tree ingests data at a much higher rate than its closest ancestor, the PDC-tree. Several strategies for mapping keys to Hilbert indices are investigated. Using the fastest Hilbert mapping for ingestion, the Hilbert PDC-tree sustains well over 350 thousand inserts per second, over 17 times faster than the PDC-tree. It also provides good

IDEAS '16, July 11 - 13, 2016, Montreal, QC, Canada

ISBN 978-1-4503-4118-9/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2938503.2938549

performance for large aggregate queries. On 50 million elements from the TPC-DS data set, using the fastest Hilbert mapping for querying, the Hilbert PDC-tree executes queries that aggregate over 66% of the data more than 4 times faster than the PDC-tree. As the number of dimensions increases, the Hilbert PDC-tree matches the good scalability of the PDC-tree for querying, and shows significantly better scalability for data ingestion. In particular, the Hilbert PDC-tree scales well to many more dimensions than R-tree variants can efficiently support.

2. BACKGROUND

2.1 Related Work

The R-tree [7] is the classical data structure for indexing multi-dimensional data sets. Over the years, numerous data structures based on the R-tree have been introduced which improve on some aspect of performance, often at the cost of another.

Many such structures are refinements to the insertion algorithm which improve query performance at the cost of insertion performance. Among the most well-known of these is the R*-tree [1][2], which occasionally re-inserts records to maintain a better structure for querying.

Others change the tree representation itself. The CR-tree [10] uses a compressed MBR representation to pack more information in a single tree node, for more efficient use of memory. An appropriate R-tree-like structure is an effective choice for indexing data sets with few dimensions, but all suffer from performance degradation when the number of dimensions gets large. The X-tree [3] improves this situation somewhat by using an overlap-minimizing split algorithm, and introducing *supernodes* with a higher fanout than usual if a good split can not be found. However, the performance problems of R-trees with many dimensions are inherent to the use of MBRs in a tree structure. As the number of dimensions increases, overlap becomes more problematic, forcing aggregate queries to search a larger portion of the tree.

To overcome this problem, the DC-tree [6] introduced the use of MDSs, which can describe many non-overlapping regions in a single key. Since the values in a dimension in an MDS are not necessarily contiguous, the DC-tree is much less prone to overlap as the number of dimensions increases. Query performance thus scales much better as the number of dimensions increases compared to MBR-based trees. The PDC-tree [5] improves overall performance in a multi-core environment, allowing correct use of the tree by many threads at once. For good speedup, a minimal locking scheme ensures threads are only blocked for a short period of time. Aside from performance, DC-trees target a slightly different problem domain than R-trees: they support hierarchical dimensions, and fast aggregation of large fractions of the data stored in the tree.

The downside of the DC and PDC trees is that working with MDSs is more expensive than MBRs. As with R-trees, many region calculations must be performed when modifying the tree, and these calculations are a bottleneck on insertion. The Hilbert R-tree [9] avoids these calculations by instead inserting records according to the Hilbert index of their keys. Since this is a linear ordering, the insertion algorithm itself is significantly faster, much like that of a B+-tree [4].

2.2 Dimension Hierarchies

DC-trees are designed to support hierarchical dimensions with discrete values, like those shown in Figure 1. Dimensions are not necessarily ordered, and a query can be at any level in each dimension. Queries specify, for each dimension, a set of values from the respective dimension hierarchy, or a wildcard indicating that the entire range of the dimension should be included. For example, a query might aggregate all sales of items with a particular brand on a particular day, in any store location, at any time of day.

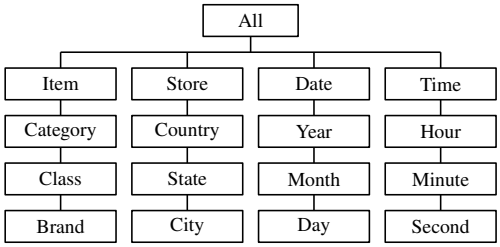


Figure 1: Some hierarchical dimensions for sales from the TPC-DS data set.

2.3 Minimum Describing Subsets

R-trees use MBRs. An MBR is simply a multi-dimensional rectangle that encloses a set of points. MBRs are limited to describing a contiguous range in each dimension, which can result in boxes that describe a much larger area than is required to enclose the relevant points.

An MDS can reduce this problem by describing a set of values in a given dimension. Figure 2 illustrates the difference between an MBR and MDS that enclose the same points. The region described by a minimal MDS is always smaller than the region described by a minimal MBR that encloses the same points, though the description of the MDS itself may be larger since it lists individual values rather than ranges.

The ability to describe several ranges in a dimension with an MDS is what makes DC-trees less susceptible to overlap as the number of dimensions increases. For example, consider the regions shown in Figure 2 as keys for directory nodes in the tree. If a value with $A = 1000$ were inserted into this subtree, the MBR's volume would increase dramatically to cover the entire range from $A = 8 \ldots 1000$. However, the MDS would only add a thin slice for $A = 1000$.

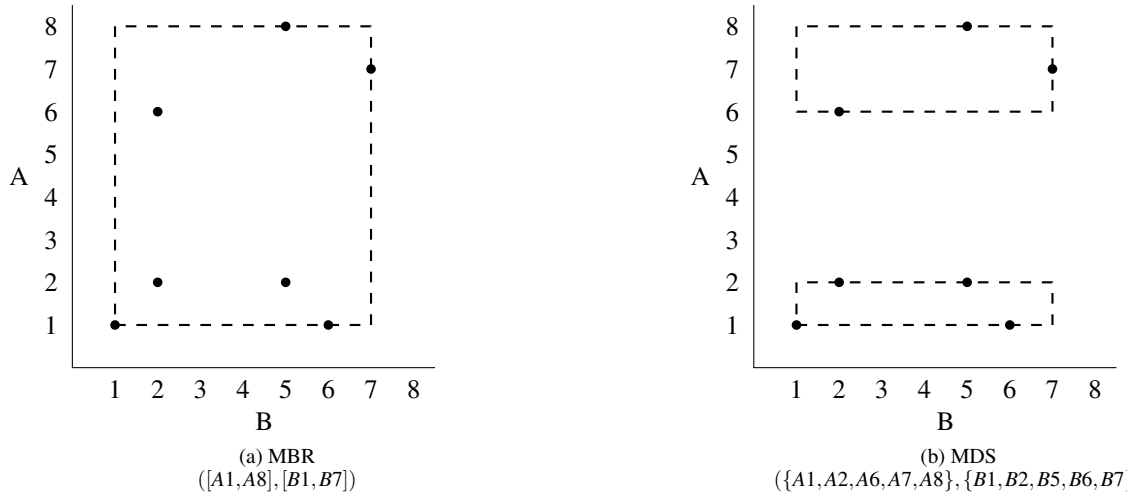
2.4 DC-tree

The DC-tree is a data structure that supports aggregate queries on data with dimension hierarchies, with many more dimensions than can be efficiently supported by R-trees. It achieves this by replacing the use of MBRs with MDSs, and using MDS-specific insertion and split algorithms designed to minimize overlap.

Values in a dimension hierarchy are stored in MDSs as integer IDs, which provides a partial ordering for IDs in the same dimension. The DC-tree uses 32-bit IDs where the level is stored in the most significant bits, and the remaining space is used as an ID within that level of the dimension, as shown in Figure 3. The dimension of an ID is known implicitly by storing each dimension in an MDS separately. Dictionaries store the parent of each value in the dimension hierarchy, allowing IDs to be converted to higher levels at the cost of looking up values in auxiliary data structures.

Figure 3: IDs in the DC-tree.

Though the details differ, at a high level the DC-tree operates much like an R-tree. Insertion locations and the distribution of children during node splits are determined by geometric comparisons, such as volume, expansion, and overlap. Aggregate queries



(a) MBR
$([A1, A8], [B1, B7])$

(b) MDS
$(\{A1, A2, A6, A7, A8\}, \{B1, B2, B5, B6, B7\})$

Figure 2: Minimum Bounding Rectangle (MBR) and Minimum Describing Subset (MDS) for the same 7 data points in 2 dimensions.

descend from the root, visiting any children that intersect the query region. To support fast aggregation of large fractions of the data, internal tree nodes contain a cached aggregate of all values in the subtree rooted at that node. Thus, if the node's key is completely contained by the query region, there is no need for the query to descend to any children of that node.

The PDC-tree is a parallel evolution of the DC-tree which uses a minimal locking scheme to allow several threads to manipulate and query the tree at once. The fundamental algorithms are similar to those of the DC-tree, but allow for speedup to be achieved in a multi-core environment.

3. HILBERT PDC-TREE

The Hilbert PDC-tree uses the same query algorithm as the PDC-tree, which is conceptually similar to the classical R-tree query algorithm, but with multi-threading, cached aggregates, and MDSs instead of MBRs.

The insertion algorithm, however, is fundamentally different. Like the Hilbert R-tree [9], the Hilbert index of keys is used to quickly determine an insert position. Each directory node in the tree stores the maximum Hilbert index of its child nodes, recursively. First, the Hilbert index of a point to be inserted is calculated. Then, starting at the root, this value is used to find the child with the largest Hilbert index not less than the new point's. This process continues until the bottom level of the tree is reached, where the new point is inserted. This is essentially the same insertion algorithm used by linearly ordered trees such as the B+-tree. Since this linear search process is much less expensive than R-tree-like geometric comparisons, insertion speed is significantly improved. Despite this much less costly insertion algorithm, the good query performance of the PDC-tree with a high number of dimension is preserved, as confirmed by the experiments shown in Section 4.

However, the Hilbert PDC-tree is not merely a straightforward application of this idea to the PDC-tree. Techniques specific to the hierarchical nature of DC-trees are employed, a new overlap-minimizing split algorithm improves query performance, and a more efficient ID and MDS representation significantly reduces overhead in general.

3.1 ID and MDS Representation

Unlike the DC-tree, the Hilbert PDC-tree does not store the level of IDs in the most significant bits. Instead, the dimension number is stored, followed by the index for the ID at each level, as shown in Figure 4. This allows IDs to be compared directly at any level by simply masking bits, without the need to consult dimension hierarchies stored elsewhere.

Dimension	Level 1 ID	Level 2 ID	Level 3 ID	Level 4 ID

Figure 4: IDs in the Hilbert PDC-tree.

The MDS structure exploits the partial ordering provided by this representation. All IDs in an MDS are stored in sorted order. This allows two MDSs to be scanned in tandem while comparing to determine if a query encloses, partially intersects, or does not touch, the MDS of a tree node. The comparison algorithm is similar to the classical linear algorithm for finding the intersection of two sorted arrays of integers, but only reports the above-mentioned cases rather than constructing a result set. However, special consideration must be made for the fact that dimensions must be compared independently, since the intersection of interest is geometric.

In order to compare two MDSs, their respective levels must be considered. IDs in a given dimension can only be compared at the same level in that dimension hierarchy. This is an issue because a query MDS may be at different levels than node MDSs encountered during the search. To address this, the DC-tree's query algorithm first adapts each MDS to comparable levels. However, looking up parent IDs and constructing new MDSs in this way is very expensive. Instead, the Hilbert DC-tree exploits its self-contained ID representation to simply "view" IDs in-place at the appropriate level. While scanning to compare MDSs, any unwanted lower levels of IDs are masked off before comparison.

3.2 Mapping Points to a Hilbert Index

Applying a Hilbert ordering to MDS keys introduces issues not present for MBRs. In particular, MDSs in the tree are expressed at various levels, where nodes higher in the tree are likely to have keys at higher levels in the dimension hierarchy. This means that MDSs are often compared at levels different than the leaf levels for which the Hilbert indices stored in the tree were calculated. Since the breadth of various levels may vary considerably across dimensions, the Hilbert order for leaves may not provide good locality for keys higher in the tree which are expressed at higher levels in the dimension hierarchy.

These issues become more important when memory consumption is taken into consideration. The high insertion rate of the Hilbert PDC-tree comes at the cost of having to store a Hilbert index for every node in the tree. With a naïve implementation, this index would take up the same amount of space as the key for a point stored in a leaf node (d integers the same size as IDs, where d is the number of dimensions). Since dimensions can have different breadths, IDs in a dimensions may not require all the available bits (e.g. 32 or 64). Using compact Hilbert indices [8] would save space by using only the minimum number of bits required, but could impact locality when MDSs are viewed at different levels.

There are several different ways to map a hierarchical point to a Hilbert index. The simplest, as shown in Table 1, map the ID directly or simply mask off the most significant bits which represent the dimension. Note that the Hilbert mapping algorithm implicitly knows the dimensions of its input values, so this value is unnecessary.

	Dim	Level 1	Level 2	Level 3	Level 4
ID	01	1	11	111	1111
	10	11	1	1	11
Direct	01	01	0011	0111	1111
	10	11	0001	0001	0011
Dimensionless	00	01	0011	0111	1111
	00	11	0001	0001	0011

Table 1: A hierarchical ID, and the simple "dimensionless" mapping which zeroes the dimension bits but preserves all level bits. Unused bits in the original ID are shown blank, but are represented by zeroes in practice.

These simple mappings have the issue that levels in different dimensions may not have the same breadth, and thus not map to a comparable range in the Hilbert ordering. The more sophisticated mappings shown in Table 2 *spread* or *expand* values in a given level to match the range of that level in any dimension. The spread mapping simply left-pads levels with zeroes to occupy the number of bits required for that level in any dimension. The expanded mapping uses the same number of bits, but shifts values left and right-pads with zeroes if necessary, ensuring values in any dimension have a similar range.

For example, in the example shown here, dimension 1 uses four bits at level 4, but dimension 2 uses only two. To compensate, values in dimension 2 at level 4 are shifted left by two bits, causing values to span roughly the same numerical range as those in dimension 1. This transformation is only performed on a copy of the key used to calculate the corresponding Hilbert index. The keys in the tree used for comparison during querying are unmodified.

	Pad	Level 1	Level 2	Level 3	Level 4
Spread	00000	01	11	111	1111
	00000	11	01	001	0011
Expanded	00000	10	11	111	1111
	00000	11	10	100	1100

Table 2: Mappings with an equal number of bits per level across dimensions.

These mappings attempt to preserve the hierarchical nature of IDs while reducing space. An alternative approach, as shown in Table 3, is to simply eliminate all unused bits and compress IDs as much as possible to minimize space, ignoring any cross-dimension hierarchical considerations.

	Pad	Levels
Compressed	000000	1111111111
	000000	0011111111

Table 3: Compressed mapping which does not preserve level sizes across dimensions.

The number of bits used by each mapping on a TPC-DS dimension hierarchy at scale 10 is shown in Figure 5.

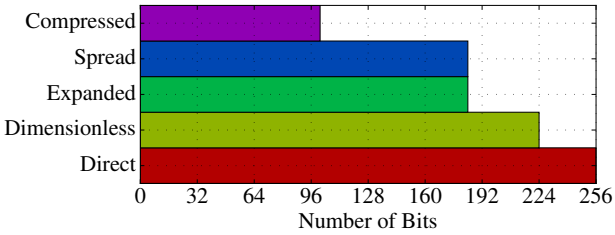


Figure 5: Number of bits used for various Hilbert mappings.

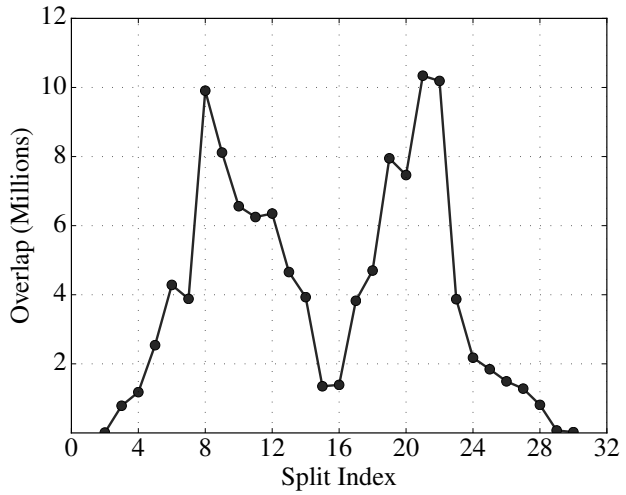
3.3 Splitting Directory Nodes

The Hilbert R-tree simply splits nodes in half like a B-tree. This achieves good balance, but at the cost of increased overlap, since choosing to split in the middle may result in significantly higher overlap than choosing to split at some other index. For a disk-based structure with few dimensions, prioritizing balance improves space utilization, so for some applications this is a reasonable choice. However, for an in-memory structure with many dimensions, high overlap degrades performance significantly more than node balance, particularly for applications that perform large aggregations.

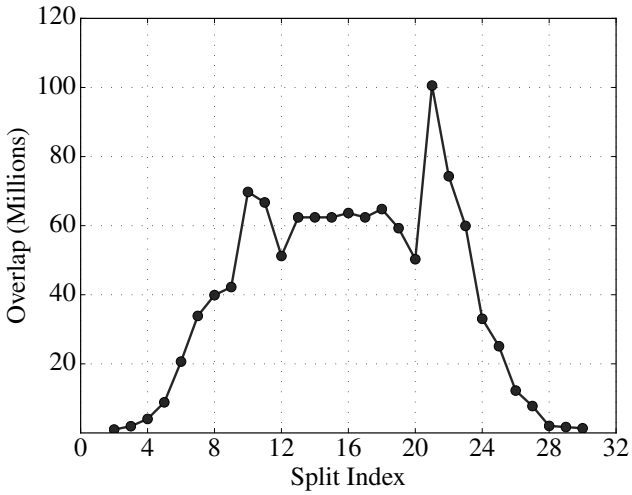
Since nodes in the tree have a linear ordering imposed by the Hilbert mapping, the overlap that would result from splitting at a given index i is simple to calculate: if L_i is the union of all boxes to the left of i, and R_i is the union of all boxes to the right of i, then the overlap that would result from splitting at index i is simply $\text{overlap}(L_i, R_i)$.

An analysis of the tree structure resulting from loading various data sets makes it clear why always splitting nodes in half can be a poor strategy for applications where overlap is critically important. Though occasionally the half-way point is a good split, as in Figure 6a, many nodes are more difficult to split well, such as that shown in Figure 6b. Here, an equal split has high overlap, but nearby indices 12 and 20 have less overlap with little cost in balance. Depending on how much balance one is willing to sacrifice for overlap, splitting at index 24 or even further may be a better choice. It is common for significantly better (often including overlap-free) splits to be possible at indices close to, but not exactly at, the fully balanced split.

By introducing a parameter for the minimum split size, the split algorithm can instead examine the overlap at each valid position and choose the one with the least overlap. Ties are broken by choosing the index with the best balance, or smallest resulting key representations, in that order. The result of using such a split algorithm with a fixed fanout of 32 are shown in Figure 7. An equal split (i.e. a split at index 16) is the most common single choice, but other indices prove better in the majority of cases. The total overlap that results from splits taken at each index are shown in Figure 7b. It is clear that the algorithm has been forced to make some poor splits. Nodes with a flat "overlap surface" contribute to the peak at the evenly balanced index, since in this case, the index with the best balance wins.



(a) A node with a good, but sub-optimal, split.



(b) A node with no clearly ideal split.

Figure 6: Overlap at each split point in observed directory nodes.

The use of *supernodes*, which have higher than usual fanout, can improve this situation. Though the split algorithm used in the Hilbert PDC-tree is significantly different, the same principle originating with the X-tree can be applied: if a split with suitably low overlap can not be found, then avoid splitting and increase the node size beyond the normal fanout to create a supernode.

The X-tree and its direct descendants do not limit the maximum size of supernodes, since a linear scan of even a very large supernode is less expensive than descending to several overlapping child nodes. However, in a parallel context, extremely large nodes harm performance since the increased locking and copying overhead limits speedup. Accordingly, the Hilbert PDC-tree adds a maximum fanout parameter which limits the size of a supernode. If a node reaches this size, it is split regardless of the quality of the best possible split. Though this introduces overlap, the addition of supernodes significantly decreases overlap even with the maximum fanout set to a small multiple of the normal fanout. Figure 8 shows the split frequencies and corresponding total overlap for a tree with maximum fanout of 128 (4 times the normal fanout of 32). Comparing with Figure 7, the distribution of split positions is very similar, with a peak at the balanced split point of 16. However, supernode splits well beyond the normal fanout range can be seen. Despite the relatively low number of supernode splits, the resulting overlap shown in Figure 8b is considerably lower than without supernodes as shown in Figure 7b.

The best split position can be found in linear time by scanning each potential split index in order. The fitness of two split positions can be compared based on overlap, balance, and key size. To compute the overlap and key size, the left and right MDSs that would result from a split at that index are required. Let $\mathscr{B}$ be the set of n MDSs corresponding to the child nodes. First, an array $\mathscr{R}$ of the right side MDSs is computed, where $\mathscr{R}[i] = \cup(\mathscr{B}[i \ldots n-1])$. This is done in $\Theta(n)$ steps by starting with $\mathscr{R}[n-1] = \mathscr{B}[n-1]$ and working left, setting $\mathscr{R}[i] = \cup(\mathscr{B}[i], \mathscr{R}[i+1])$.

The algorithm proceeds left to right, similarly computing the left-hand MDS l as it proceeds. For each split index i, l and $\mathscr{R}[i+1]$ can be used to compute the overlap and key size that would result from a split at i. The balance is trivially computed from i and the tree fanout. Each potential split is evaluated according to these criteria, and if a better split is found, the best split index is updated. However, a split index is only considered "best" if it is a local minimum with respect to overlap, or has zero overlap. This restriction avoids the degenerate case of always choosing the first or last index within the minimum split size in situations like the one shown in Figure 6a. In cases where no local minima are found, it is likely that a better split would be found outside the current range being considered, so avoiding a split and making a supernode will likely result in a better split in the future.

4. EXPERIMENTS

To evaluate the performance of the Hilbert PDC-tree, experiments were performed using the TPC-DS [11] data set with scale factor 50 on an Intel Core i7-3770 (4 cores, 8 threads) with 32 GiB of RAM. These experiments use 8 hierarchical dimensions: `item`, `store`, `sold_date`, `sold_time`, `addr`, `cdemo`, `hdemo`, and `customer`.

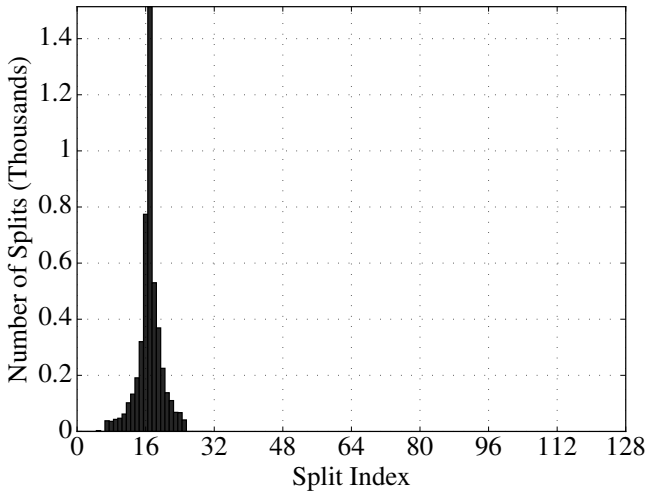
Six tree variants are tested: the five Hilbert mappings described in Section 3.2, and the PDC-tree. This PDC-tree uses the same underlying implementation as the Hilbert PDC-tree, including the exact same query code, so the comparison is fair in terms of implementation efficiency. Each tree is evaluated on streams consisting of only inserts, only aggregate queries, and a 50% mix of inserts and aggregate queries. The latencies shown are the average for many operations, where each individual test at a given size is run until at least 20 queries are executed and at least 4 seconds have passed.

Query *coverage* is the fraction of the data included in the query result. For example, a query with 50% coverage aggregates half of the items stored in the tree. Test queries are randomly generated before benchmarking, and placed into bins according to their coverage. The generated queries include several possibly non-contiguous values at different levels in all dimensions. Queries range from very low coverage to aggregations of nearly the entire data set.

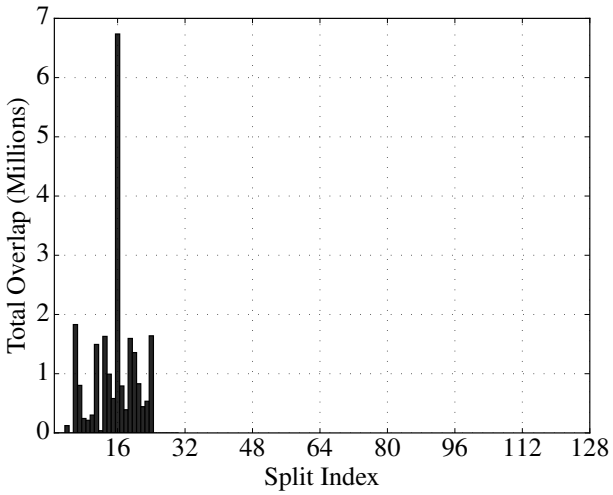
Fanout is configured to 32, with a maximum fanout of 128 for supernodes. Except where explicitly shown, the values shown are for 8 threads, which showed the best overall performance on the hardware used.

4.1 Data Ingestion

Performance for a stream that contains only insertions is shown in Figure 9. It is clear that the Hilbert PDC-tree achieves a much higher rate of ingestion than the PDC-tree. The various Hilbert

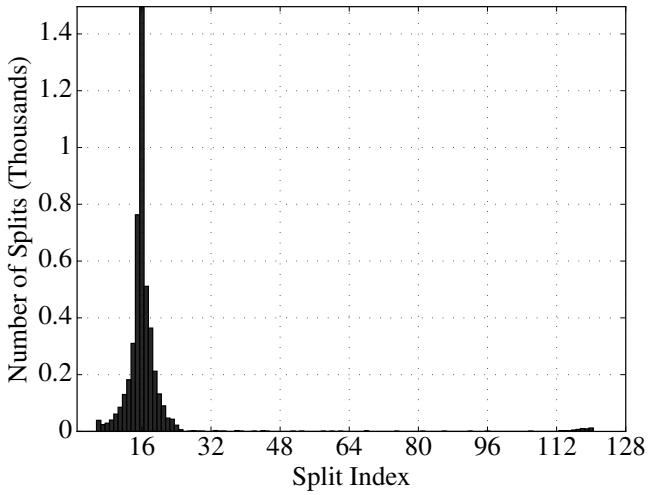


(a) Distribution of split positions

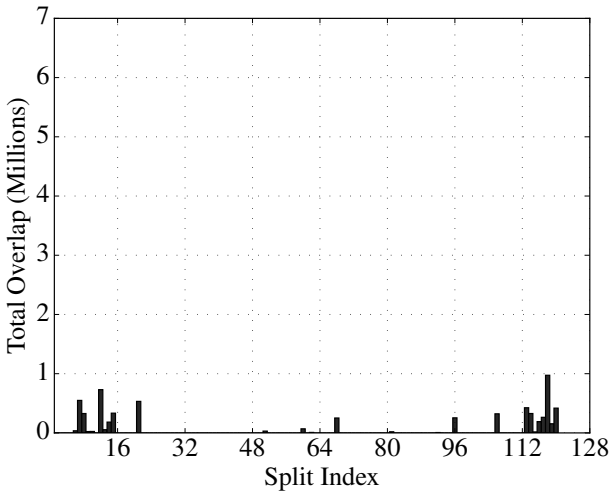


(b) Total resulting overlap

Figure 7: Split index frequency and overlap with fixed maximum fanout.



(a) Distribution of split positions



(b) Total resulting overlap

Figure 8: Split index frequency and overlap with supernodes.

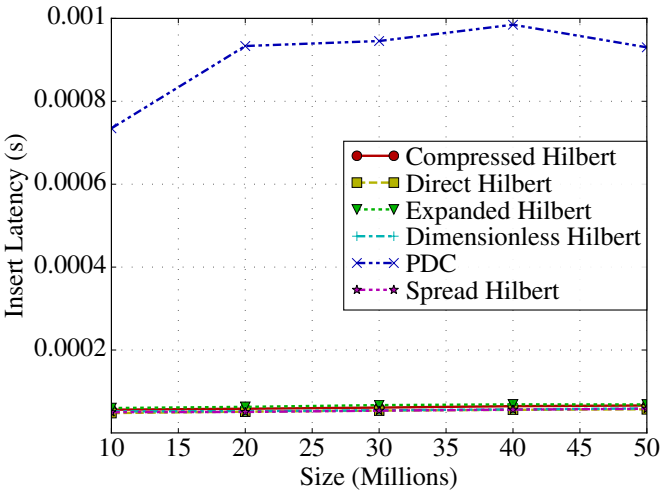
mappings range in throughput by about 100 thousand inserts per second, but even the slowest (expanded) at approximately 275 thousand inserts per second is over 13 times faster than the PDC-tree at approximately 20 thousand inserts per second. The fastest Hilbert mapping for ingestion (direct) sustains well over 350 thousand inserts per second, over 17 times faster than the PDC-tree. This performance edge is maintained as data size increases, with all trees showing a relatively flat trend.

The performance discrepancy between the various Hilbert mappings is mainly due to the expense of manipulating the keys before applying the Hilbert mapping. The direct mapping is the fastest, since no copying or manipulation is required at all. The expanded and compressed mappings are slower since more bit shifting is required, as well as dictionary lookups to find the appropriate range of bits for a level. Further optimization of the Hilbert mapping could reduce this gap and improve overall ingestion performance, but the cost of Hilbert mapping is very low compared to the geometric comparisons performed by the PDC-tree.

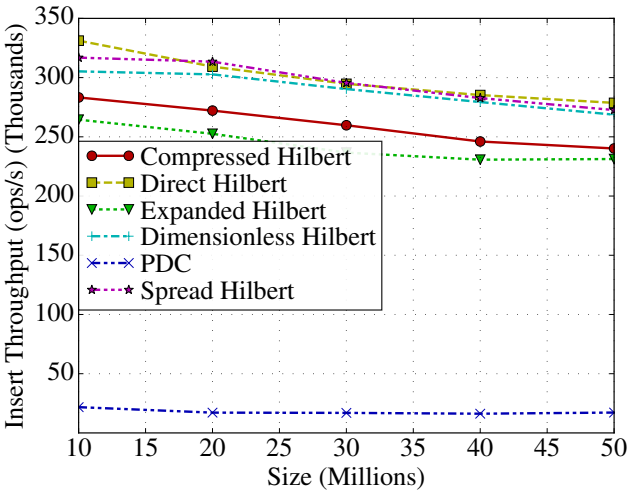
4.2 Querying

Performance for a stream that contains only aggregate queries is shown in Figure 10. The difference in performance between the various Hilbert mappings is apparent, with the expanded mapping performing better than all other configurations. The expanded mapping performs best because it spreads each level of each dimension evenly throughout the Hilbert space, regardless of the range actually covered by the data. This results in a tree structure that is more resilient to the order in which elements are inserted. However, this mapping is also the slowest during ingestion, so there is a slight trade-off between insert and query performance where the best choice may differ depending on application.

It is particularly apparent that the compressed mapping, which uses a minimal amount of space for Hilbert indices, performs poorly. This is because the compressed mapping does not preserve the hierarchical nature of IDS at all, so locality is not well-preserved with higher level keys which is particularly significant for large aggregations.



(a) Insert Latency



(b) Insert Throughput

Figure 9: Performance with a stream of inserts.

Compared to the PDC-tree, all Hilbert PDC-tree variants except the compressed mapping perform considerably better, though the improvement is not as dramatic compared to insertion. This is in line with the design goals of the Hilbert PDC-tree: increase the rate of data ingestion, while preserving the good query performance of the PDC-tree.

Performance for a mixed stream of 50% inserts and 50% queries is shown in Figure 11. These timings are very similar to the query-only timings shown in Figure 10, since the overall time is dominated by queries which take much more time to execute than insertions in general. Performance is slightly better than the query-only workload, which confirms that the minimal-locking scheme inherited from the PDC-tree keeps thread contention low, preventing concurrent inserts from significantly harming query performance.

A breakdown of the best Hilbert PDC-tree variant (expanded) and the PDC-tree performance with respect to query coverage is shown in Figure 12. These figures are for a stream of only queries, chosen from 3 bins (one for each coverage range) uniformly at random. For this dataset, the Hilbert PDC-tree out-performs the PDC-tree for all query coverages. The best improvement is seen for queries with medium to high coverage, since a good tree structure allows high coverage queries to better utilize cached aggregate values and avoid traversing many large subtrees.

4.3 Speedup and Hilbert Mappings

The throughput speedup for a mixed stream of 50% inserts and 50% aggregate queries is shown in Figure 13. The best speedup occurs when using 8 threads, which corresponds to the number of hardware-supported threads on this processor. Speedup for the best mapping, expanded, peaks at just under 4, the number of physical cores present. Increase in speedup roughly corresponds to the query performance seen by each tree variant. This is expected, since query time dominates, and a more efficient tree structure results in each thread visiting fewer nodes, which reduces potential thread contention that would limit speedup.

The benefits of the various Hilbert mappings depend on several factors, including data distribution and application requirements. However, in general, the expanded mapping is typically best where query performance is the primary concern. In applications where ingestion throughput has top priority, the spread mapping is the best choice, supporting a significantly higher rate of ingestion with only a modest penalty in query performance.

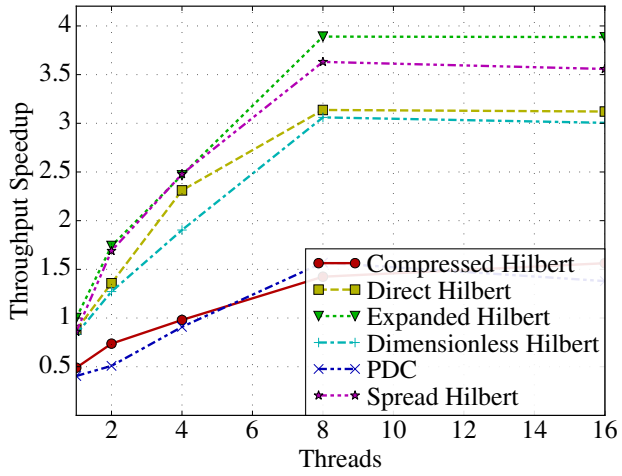
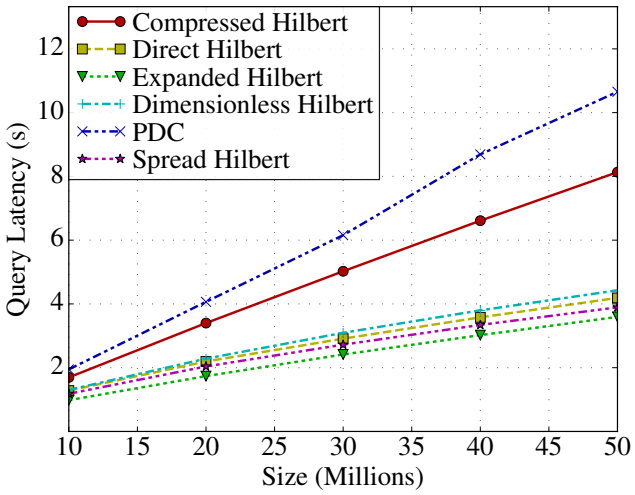


Figure 13: Speedup for a mixed stream of inserts and aggregate queries.

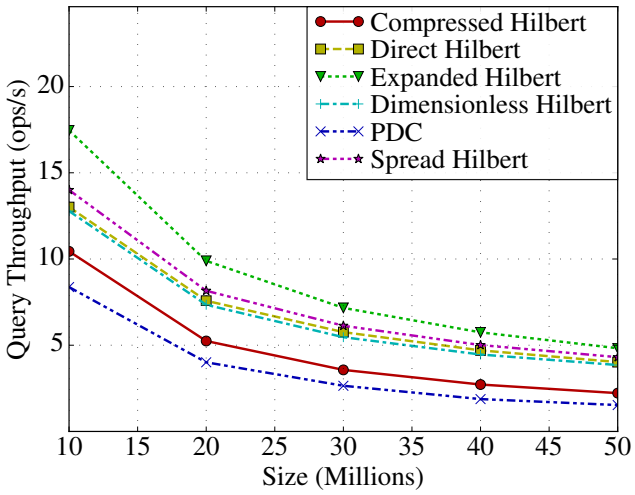
4.4 Dimensionality

To evaluate how the Hilbert PDC-tree scales as the number of dimensions increases, experiments were run using synthetic Zipf-distributed data with moderate skew (0.5). For comparison, two R-tree variants were included: one using the classic insertion algorithm (R-tree), and the other using Hilbert ordering (Hilbert R-tree). All trees are based on the same underlying parallel tree implementation and benefit from the same optimizations.

The various trees are compared by executing the same queries. Since the generated test queries are MDSs that may include non-contiguous values, it is not possible to directly convert queries to a single MBR for testing R-tree variants. Instead, each query MDS is converted into several MBRs, and the total time to execute all the resulting MBR queries is measured. This compares the trees fairly by performing an equivalent task, that is, the results obtained from each tree are identical.

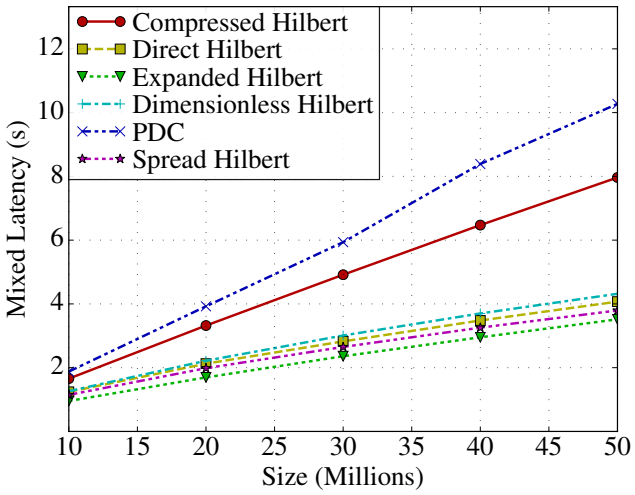


(a) Query Latency

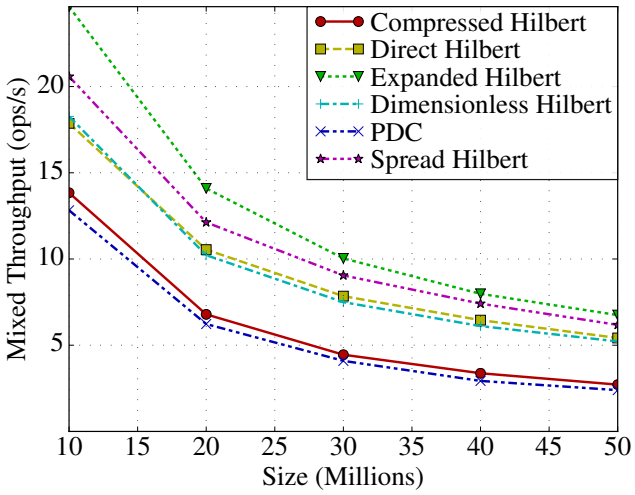


(b) Query Throughput

Figure 10: Performance for a stream of queries.



(a) Mixed Latency



(b) Mixed Throughput

Figure 11: Performance for a mixed stream of 50% inserts and 50% queries.

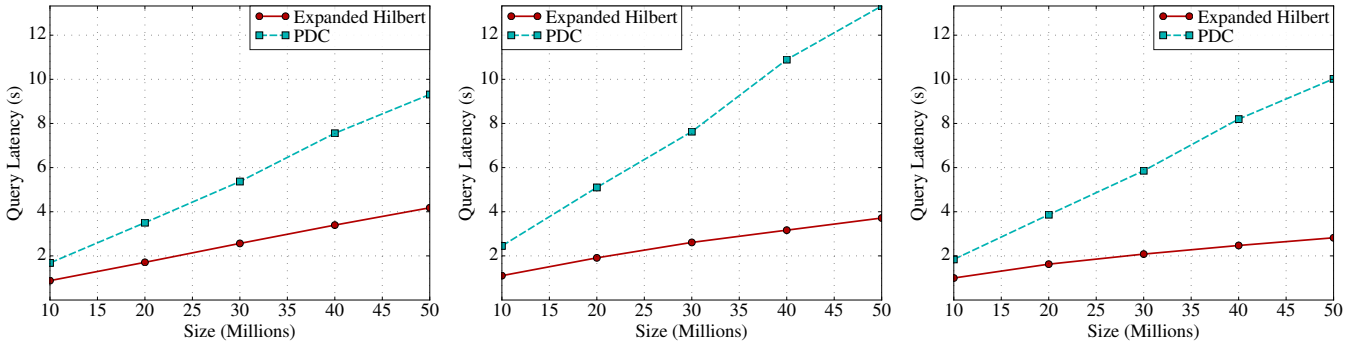
With few dimensions, the R-tree variants perform better since MBRs are less expensive to work with. However, as the number of dimensions increases, the benefits of the MDS-using PDC-trees become clear. Past 8 dimensions, the query performance of the R-tree variants degrades dramatically. Queries beyond 20 dimensions take an unreasonably long time to complete for the R-tree variants, so these experiments were stopped at this point.

The Hilbert PDC-tree preserves the PDC-tree's ability to gracefully scale to a large number of dimensions, though the query performance advantage seen in the (8 dimension) TPC-DS results narrows as the number of dimensions increases. However, the improvement in insert performance is not only preserved, but becomes greater with more dimensions. This is because the the PDC-tree must perform more expensive geometric comparisons as the number of dimensions increases, but the insertion algorithm for the Hilbert PDC-tree is based on a linear ordering. As the number of dimensions increases, the cost of mapping keys to Hilbert indices and the cost of comparing Hilbert indices increases slightly, but this overhead is very small compared to the additional work the PDC-tree must perform. Consequently, the Hilbert PDC-tree shows almost no increase in insert latency as the number of dimensions increases, but inserting into the PDC-tree becomes significantly more expensive.

5. CONCLUSION

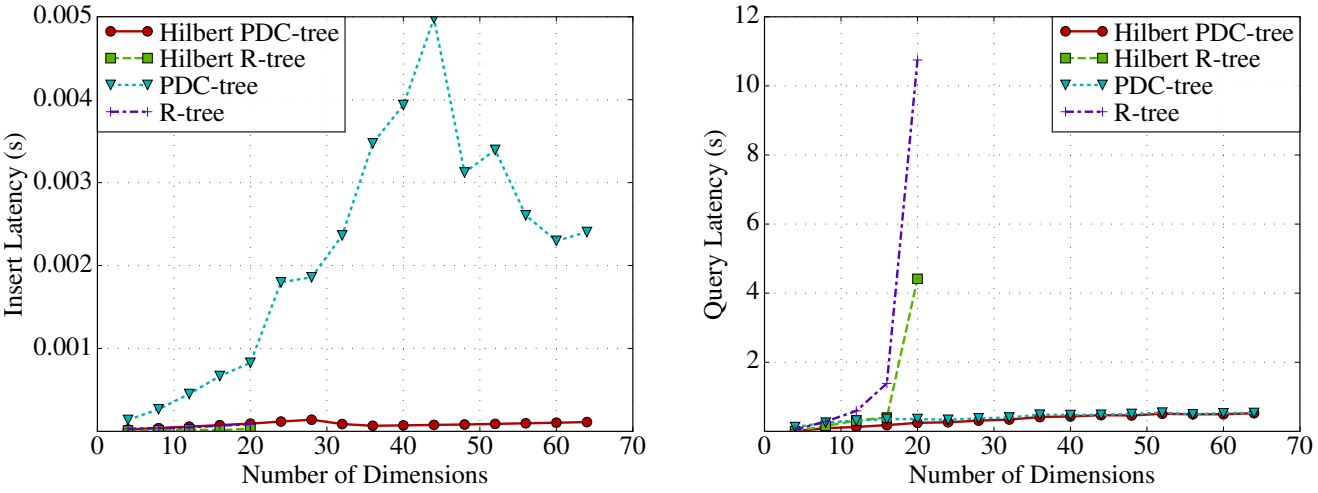
The Hilbert PDC-tree is a parallel index for many-dimensional data that supports a high rate of data ingestion. This is achieved by avoiding geometric comparisons during insertion by instead inserting items based on the Hilbert index of their keys. A new overlap-minimizing node split algorithm maintains a structure that efficiently supports aggregate queries.

Experiments confirm that the Hilbert PDC-tree ingests data at a much higher rate than its closest ancestor, the PDC-tree, while providing good performance for queries that aggregate large fractions of the data stored in the tree. In particular, the Hilbert PDC-tree scales well to many more dimensions than R-tree variants can efficiently support.



(a) Low coverage (< 33%) (b) Medium coverage (c) High coverage (> 66%)

Figure 12: Aggregation query latency for various coverages.



(a) Insert latency (b) Query latency

Figure 14: Latency as number of dimensions is increased.

Acknowledgment

Research partially supported by the IBM Center for Advanced Studies Canada and the Natural Sciences and Engineering Research Council of Canada.

6. REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. *ACM SIGMOD Record*, 19(2):322–331, May 1990.

[2] N. Beckmann and B. Seeger. A revised R*-tree in comparison with related index structures. In *Proc. 2009 SIGMOD Int. Conf. on Management of Data*, page 799. ACM, 2009.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22th Int. Conf. on Very Large Data Bases*, pages 28–39. Morgan Kaufmann.

[4] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, Jun 1979.

[5] F. Dehne and H. Zaboli. Parallel real-time OLAP on multi-core processors. In *Proc. 12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, pages 588–594, 2012.

[6] M. Ester, J. Kohlhammer, and H.-P. Kriegel. The DC-tree: A fully dynamic index structure for data warehouses. In *Proc. 16th Int. Conf. on Data Engineering*, pages 379–388. IEEE Comp. Soc., 2000.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14(2):47–57, 1984.

[8] C. H. Hamilton and A. Rau-Chaplin. Compact Hilbert indices: Space-filling curves for domains with unequal side lengths. *Information Processing Letters*, 105(5):155–163, Feb 2008.

[9] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 500–509, 1994.

[10] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proc. 2001 SIGMOD Int. Conf. on Management of Data*, pages 139–150. ACM, 2001.

[11] M. Poess, B. Smith, L. Kollar, and P. Larson. TPC-DS, Taking decision support benchmarking to the next level. In *Proc. 2002 SIGMOD Int. Conf. on Management of Data*, pages 582–587. ACM, 2002.