Towards a Painless Index for Spatial Objects †

RUI ZHANG, JIANZHONG QI, MARTIN STRADLING, and JIN HUANG
The University of Melbourne

Conventional spatial indexes, represented by the R-tree, employ multi-dimensional tree structures, which are complicated and require enormous efforts to implement in a full-fledged database management system (DBMS). An alternative approach for supporting spatial queries is mapping based indexing, which maps both data and queries into a one-dimensional space such that data can be indexed and queries can be processed through a one-dimensional indexing structure such as the B^+-tree. Mapping based indexing requires implementing only a few mapping functions, incurring much less efforts in implementation compared to conventional spatial index structures. Yet, a major concern about using mapping based indexes is their lower efficiency than the conventional tree structures.

In this paper, we propose a mapping based spatial indexing scheme called *Size Separation Indexing* (SSI). SSI is equipped with a suite of techniques including size separation, data distribution transformation, and more efficient mapping algorithms. These techniques overcome the drawbacks of existing mapping based indexes and improve the efficiency of query processing significantly. We show through extensive experiments that for window queries on *spatial objects with non-zero extents*, SSI has two orders of magnitude better performance than existing mapping based indexes and has competitive performance to the R-tree as a standalone implementation. We have also implemented SSI on top of two off-the-shelf DBMSs, PostgreSQL and a commercial platform, both of which have R-tree implementation. In this case, SSI is up to two orders of magnitude faster than their provided spatial indexes. Therefore, we achieve a spatial index which is more efficient than the R-tree in a DBMS implementation and at the same time easy to implement. This result may upset a common perception that has existed for a long time in this area that the R-tree is the best choice for indexing spatial objects.

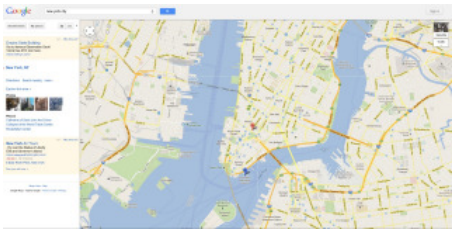
Categories and Subject Descriptors: H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Spatial databases, mapping based indexing, window queries, space-filling curves

†There is a patent pending on this technique; the code for this technique is available through http://www.ruizhang.info/code.htm

Authors' address: Rui Zhang, Department of Computing and Information Systems, University of Melbourne, VIC 3010, Australia email: rui.zhang@unimelb.edu.au; Jianzhong Qi, Department of Computing and Information Systems, University of Melbourne, VIC 3010, Australia email: jianzhong.qi@unimelb.edu.au; Martin Stradling, Department of Computing and Information Systems, University of Melbourne, VIC 3010, Australia email: martin.stradling@gmail.com Jin Huang, Department of Computing and Information Systems, University of Melbourne, VIC 3010, Australia email: jin.huang@unimelb.edu.au



(a) spatial objects within a (browser) window (b) molecular compounds within hyper-rectangles

Fig. 1. Window queries on spatial objects in real applications

1. INTRODUCTION

Spatial databases are becoming ubiquitous. They play an important role in applications such as geographic information systems (GIS), computer-aided-design (CAD), medical imaging [Arya et al. 1994], and drug design [Zauhar et al. 2003]. The recent proliferation of digital map applications, e.g., Google Maps, and location based social network services, e.g., FourSquare, is further evidence of the significance of spatial databases. A digital map may have tens of millions of objects (shops, buildings, restaurants, etc), and map service providers may need to answer millions of queries per day [Google 2012]. In these applications, one of the most basic and common operations is retrieving all objects in a rectangle. For example, Google Maps needs to display all objects in the range of a browser window (Fig. 1a). In drug design, we need to find molecular compounds that fall in certain 3D rectangular ranges (Fig. 1b). Fast object retrieval is essential to user experience in these applications. While large enterprises like Google can deploy an enormous number of computers to achieve high retrieval efficiency, it is economically prohibitive for smaller companies to do the same and they generally show less favorable performance (e.g., Whereis[1], a smaller online map service provider, needs several seconds to retrieve the restaurants in a city downtown). To ensure fast response for retrieving a large number of spatial objects with constrained computing resources, spatial index is an important approach.

One of the main reasons that stops many real applications from employing a spatial index is that spatial indexes are often too complicated to be implemented. One of the most popular spatial indexes, the R-tree, was originally proposed in 1984 [Guttman 1984], but it was not until 1994 did Oracle start to work on Oracle Spatial, and not until at least five years later that Oracle had built an enterprise-level implementation of the R-tree [Oracle 2007]. For another major DBMS provider, Microsoft, its main DBMS Microsoft SQL server only starts to have spatial support in the 2008 version [Microsoft 2008]. Practices of these major companies suggest that implementing a spatial index in a full-fledged commercial DBMS incurs significant efforts (and "pains").

The "pain" of implementing a spatial index in a DBMS motivates us to find a "painless" way to support spatial queries. Mapping based indexing is a promising direction towards this goal [Zhang et al. 2005]. It first maps spatial data objects

[1]http://www.whereis.com

into one-dimensional values and then indexes those values using a one-dimensional indexing technique, typically the B^+-tree. The B^+-tree has been implemented in most existing DBMSs. Thus, implementing a mapping based indexing requires only implementing some mapping functions (detailed in Section 2).

The major concern about using mapping based indexing is its relatively low query efficiency compared to tree based indexing. This has been the general perception among the database community. Moreover, existing studies of mapping based indexing has been focusing on point data with high dimensionality rather than general spatial objects with non-zero extents (e.g., [Berchtold et al. 1998; Jagadish et al. 2005; Zhang et al. 2004]). Very little existing work reports the effectiveness of mapping based indexing on spatial objects with **non-zero extents and low-dimensionality**.

In this paper, we reexamine the mapping based approach for indexing spatial objects with non-zero extents and focus on processing the most basic and common query, the window query. We propose key techniques to overcome the drawbacks of the mapping based indexing, namely, size separation, data distribution transformation, fast mapping functions, and a cost model for skewed data handling and parameter tuning. We call the resultant indexing scheme *Size Separation Indexing (SSI)*. From an extensive experimental study we observe that SSI has similar query efficiency to the R-tree as a standalone implementation and outperforms the R-tree significantly when implemented on top of an existing DBMS — this is because complex tree structures incur more overhead in concurrency control and transaction management in a DBMS. As a result, we achieve an efficient spatial index that is very easy to implement either through integrating into an existing DBMS or on top of it. This result may upset a common perception that has existed for a long time that the R-tree is the best choice for indexing spatial objects and processing queries upon them. The contributions of this paper are summarized as follows:

(1) We propose a new mapping based indexing scheme named Size Separation Indexing (SSI), which supports efficient window queries on spatial objects with non-zero extents. SSI consists of the following novel techniques:
a) a size separation technique to separate objects into partitions based on their size and index each partition with a space-filling curve;
b) a cumulative mapping method to map objects with skewed position distribution into a space where they are uniformly distributed;
c) two efficient algorithms to map a window query to space-filling curve ranges;
d) a cost model to optimize the parameters used in the previous steps.

(2) We show how SSI can be implemented on top of an existing DBMS easily and describe the key implementation steps.

(3) We conduct extensive experiments. The results show that:
a) SSI outperforms the best existing mapping based technique by at least five times in terms of both response time and number of page accesses.
b) when implemented as a standalone software, SSI achieves similar performance to the R-tree in most cases and better performance when the data set cardinality or the selectivity of the window query get large;
c) when implemented upon two full-fledged DBMSs, SSI significantly outperforms the R-tree implementation in these DBMSs by up to three orders of

magnitude.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 describes the preliminaries. Section 4 presents SSI, detailing how the objects are indexed and queried under SSI. Section 5 gives a cost model for optimizing SSI. Section 6 shows how SSI can be implemented on top of a DBMS. Section 7 analyzes the empirical performance of SSI and Section 8 concludes the paper.

2. RELATED WORK

In this section we provide backgrounds on querying spatial databases and review studies on indexing schemes to support spatial queries. We focus on the spatial window query (the range query) for that it is arguably the most basic and common spatial query and it can be used as a building block for other types of spatial queries such as the spatial join query and the (k) nearest neighbor query [Zhang et al. 2005].

A window query returns all objects that satisfy certain predicates (containment, intersection, etc.) with regard to a (hyper-)rectangular region called the query window. A naive algorithm to process the window query is the *scan algorithm*, which checks every object in the data set against the window query and predicate. Despite its simple idea, the algorithm can be competitive in certain cases [Weber et al. 1998] because it only accesses the data sequentially, while more complex methods may need random access on the data.

Spatial indexes are commonly used to accelerate the process of spatial queries like the window query. There are three major categories of spatial indexing schemes, namely, balanced hierarchical indexing, unbalanced hierarchical indexing, and mapping based indexing. In what follows, we discuss the three categories in more detail.

2.1 Balanced Hierarchical Indexing

A balanced hierarchical spatial index first performs *data based partitioning*, which partitions spatial objects based on their clustering relationship, and then indexes the partitions hierarchically. One of the most commonly used hierarchical spatial index is the R-tree [Guttman 1984]. From this many variants have been proposed. Among them, the R*-tree [Beckmann et al. 1990] is probably the most popular one. We use it in our experiments for performance comparison.

The R-tree is a multi-dimensional extension to the B^+-tree. It consists of a hierarchy of nested multi-dimensional intervals. Objects are represented by their minimum bounding rectangles (MBR) that are the smallest (hype-)rectangles fully containing the objects. Each R-tree node usually corresponds to a single disk page, which makes the R-tree fit nicely in a disk. To perform a window query on an R-tree, we start from the root and check which MBRs in the root satisfy the window query, and continue to search child nodes corresponding to these MBRs. It is common that multiple branches of an R-tree need to be explored during the query process due to overlaps between R-tree nodes. This is a major drawback of the R-tree.

The R*-tree improves the insertion procedure of the R-tree to reduce the overlaps between tree branches. It considers more factors like the overlaps between nodes and region perimeters. It also introduces the forced reinsertion when a node is overfull to redistribute the objects more evenly.

GiST [Hellerstein et al. 1995] is an index template implemented in PostgreSQL

which allows users to implement balanced tree indexes of domain specific data types. It can be used to implement the R-tree relatively easily. However, implementing GiST itself requires modifying the DBMS kernel, which is difficult. This means that using GiST to implement the R-tree is limited to PostgreSQL and cannot be easily applied to other widely used DBMSs such as Oracle Database, Microsoft SQL Server and IBM DB2. Also, the complex tree structure of GiST will still incur more overhead in concurrency control and transaction management.

2.2 Unbalanced Hierarchical Index Structures

An unbalanced hierarchical spatial index structure first performs *space based partitioning*, which recursively partitions the data space. Objects in the same space partition are grouped together and then indexed hierarchically. Examples of this type of index structures include the quadtree [Finkel and Bentley 1974] and the k-d tree [Bentley 1975]. The space partition based grouping leads to unbalanced trees, which makes this type of indexes difficult to fit in disks. Thus, this type of indexes is more suitable for main memory databases, and is seldom implemented in full-fledged DBMSs. Since we consider a more general case where the data sets cannot be fully held in the main memory, this type of indexes will not be discussed further in our study.

Note that SP-GiST [Aref and Ilyas 2001] provides a template for implementing unbalanced tree indexes on PostgreSQL. It manages the mapping of the nodes of an unbalanced tree to disk pages. However, SP-GiST still shares the drawbacks of GiST, while the tree node mapping makes it even more complicated.

2.3 Mapping Based Indexing

A mapping based indexing first maps multi-dimensional data into one-dimensional values and then indexes the values in a one-dimensional index structure. Three components are usually involved in a mapping based indexing.

(1) Every data object is mapped to a (or multiple) one-dimensional value(s). A one-dimensional indexing structure (e.g., the B^+-tree) is used to index the mapped values of all data objects.

(2) A (window) query in the original data space is mapped to a set of one-dimensional range queries. Objects whose mapped one-dimensional values lie in those ranges are retrieved using the one-dimensional indexing structure.

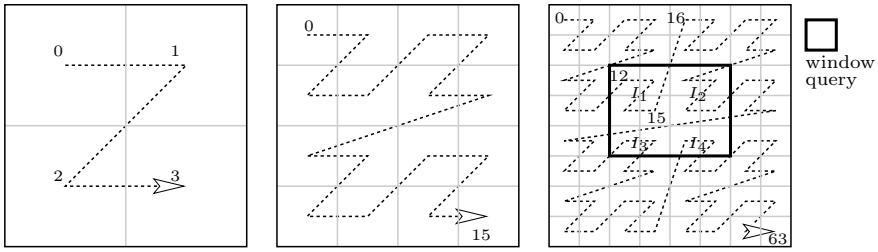
(3) The retrieved objects are checked against the query and false positives are removed. Then the final query result is returned.

Zhang et al. [2005] proposed a generalized framework named GiMP for multi-dimensional data mapping and query processing. GiMP abstracts the design of a mapping based indexing technique to the definition of several basic functions: (i) a function for data mapping, (ii) insertion and deletion functions for index maintenance, and (iii) a function for query mapping. How a mapping based indexing works is largely determined by the mapping functions. For spatial data with low dimensionality, popular mapping schemes include space-filling curves, dual-transformation [Gaede and Günther 1998] and double transformation (DOT) [Faloutsos and Rong 1991]. More recently, an index called B^{dual}-tree [Yiu et al. 2008] pro-

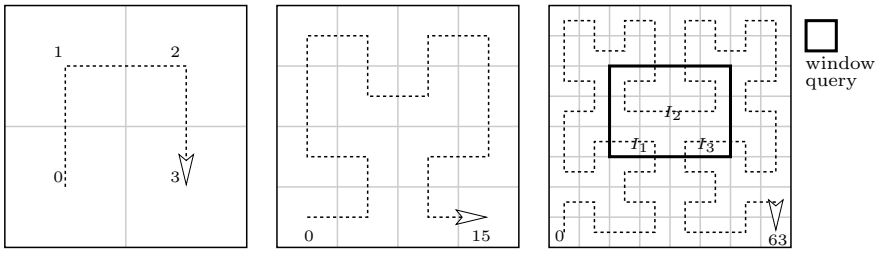
vides a mapping scheme for moving spatial objects. Next we discuss these mapping schemes in detail.

Note that there are also mapping schemes for data with higher dimensionality like Pyramid [Berchtold et al. 1998] and iDistance [Jagadish et al. 2005], but they are not the focus of this study and hence will not be discussed further.

2.3.1 *Space-filling curves.* Space-filling curves are based on a conceptual grid partitioning of the data space. A curve goes through every cell of the grid in a certain order. When the curve reaches a cell, it associates the cell with an integer, i.e., the number of cells reached so far, as the *curve value* of the cell (the association starts from cell 0). Then a multi-dimensional object can be mapped to the curve vales of the cells this object intersects.



(a) Z-curves with order $\lambda = 1, 2, 3$



(b) Hilbert-curves with order $\lambda = 1, 2, 3$

Fig. 2. Space-filling curves of different orders

Fig. 2 shows two types of popular space-filling curves, the Z-curve [Orenstein and Merrett 1984] and the Hilbert-curve [Faloutsos and Roseman 1989] with different orders. Here, the *order of a space-filling curve* is the number of times the space has been subdivided to form the conceptual grid. For example, in Fig. 2a, the left most is an order 1 Z-curve, which only divides the space once into four partitions. Then the middle one, an order 2 Z-curve, further divides each of the partitions into four sub-partitions. This process repeats and results in an order 3 Z-curve, which is shown in the right most figure.

To implement a mapping based indexing with a space-filling curve, two mapping functions [Butz 1971; Faloutsos and Roseman 1989; Ramsak et al. 2000] are needed. One is a mapping function $C(x)$ that maps a point x in the space to its curve value m. The other is an inverse mapping function $C^{-1}(m)$ that takes a curve value m and returns the position x.

To process a window query with a space-filling curve, a mapping function is needed to map the window query into intervals of curve values that represent the segments of the curve bounded by the window query. For example, in Fig. 2a, the mapping function should map the window query to four *Z-curve value (Z-value for short)* intervals I_1, I_2, I_3, and I_4. Similarly, in Fig. 2b, the mapping function should map the window query to three *Hilbert-curve value (Hilbert-value for short)* intervals I_1, I_2, and I_3. A well known mapping function for the Z-curve [Ramsak et al. 2000] uses two functions *getNextZvalue* and *getNextZvalueExit* to compute the intervals. Given a Z-value of a cell *outside* the window query, *getNextZvalue* computes the closest (next) Z-value that corresponds to a cell *inside* the window query. For example, in Fig. 2a, if 0 is fed to *getNextZvalue*, then the function will return 12. Given a Z-value of a cell *inside* the window query, *getNextZvalueExit* computes the closet (next) Z-value that corresponds to a cell *outside* the window query. In Fig. 2a, if 12 is fed to *getNextZvalueExit*, then the function will return 16. Calling these two functions repeatedly until *getNextZvalue* meets the end of the Z-curve will give us all Z-value intervals corresponding to the window query.

Lawder and King [2001] present an algorithm similar to the above one for window query mapping on Hilbert-curves. This algorithm uses the *calculate_next_match* function, which is the counterpart of *getNextZvalue* for Hilbert-curves, i.e., it computes the next Hilbert-value *inside* the window query. However, this algorithm does not have a function similar to *getNextZvalueExit* to compute the next Hilbert-value *outside* the window query. It has to access the nodes (i.e., disk pages) of a B^+-tree index on the data objects to determine the next Hilbert-value *outside* the window query. Specifically, the algorithm starts with feeding the smallest key indexed in the B^+-tree into *calculate_next_match* to find the first Hilbert-value within the window query. Then the B^+-tree node N whose key range covers this Hilbert-value is accesses to identify the objects intersecting the window query. After node N has been accessed, the largest key value in N is fed into *calculate_next_match* to identify the next B^+-tree node to be accessed. This process is repeated until there is no more B^+-tree nodes to be accessed.

We may adapt Lawder and King's algorithm as follows so that it can map the window query without accessing the B^+-tree index and hence without accessing any disk pages. Let q be the window query and $\mathcal{Z}$ be the entire data space. We use $\mathcal{Z} \setminus q$ to denote the region in $\mathcal{Z}$ that is *not* covered by q. We start the mapping with feeding 0 into *calculate_next_match* to find the first Hilbert-value within q, denoted by h_0. Then we treat $\mathcal{Z} \setminus q$ as the window query, and feed h_0 into *calculate_next_match* to find the first Hilbert-value within $\mathcal{Z} \setminus q$ that is larger than h_0, denoted by h_1. The value h_1 is also the next Hilbert-value that exists q. Now we have a Hilbert-value interval $[h_0, h_1 - 1]$ that is covered by q. We then feed h_1 into *calculate_next_match* and repeat the procedure above to identify the next Hilbert-value interval covered by q. This process continues until *calculate_next_match* meets the end of the curve, which will give us all Hilbert-value intervals enclosed by q.

We call this adapted algorithm $GetNextH$ and use it as one of the baseline algorithms in Section 7.1.3.

2.3.2 *Dual-transformation.* Dual-transformation is based on the idea that a hyper-rectangle in an n-dimensional space can be viewed as a point in a $2n$-

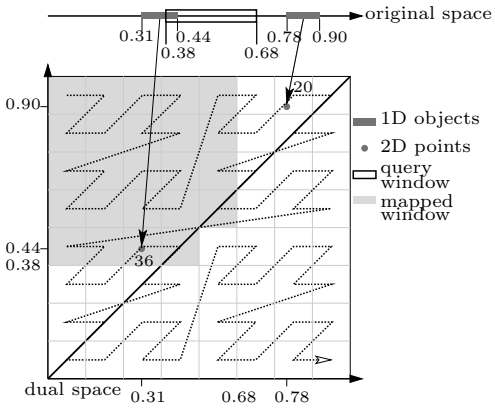


Fig. 3. Dual-transform for 1D objects

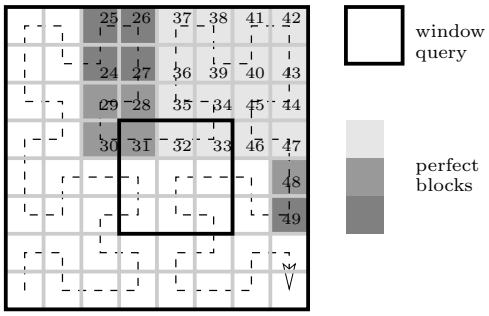


Fig. 4. Searching one node using B^{dual}-tree

dimensional space, which is then mapped to a one-dimensional space through space-filling curves and indexed with a one-dimensional index structure [Comer 1979; Faloutsos and Rong 1991; Gaede and Günther 1998]. The aim of this technique is to avoid indexing rectangles, which have non-zero extents and may be mapped to multiple curve values, while a point will only be mapped to one curve value.

Fig. 3 gives an example. Two 1-dimensional objects represented by two one-dimensional ranges $[0.31, 0.44]$ and $[0.78, 0.9]$ are transformed to two points (0.31, 0.44) and (0.78, 0.9) in a 2-dimensional space where the range bounds are used as the coordinates. The coordinates are then mapped by a Z-curve and the resultant Z-values are 36 and 20, respectively. When performing a window query $[0.38, 0.68]$, we know that an object can only intersect this window when its right bound is larger than 0.38 and its left bound is smaller than 0.68. Hence, when transformed into a 2-dimensional space, the window query should bound all cells whose bottom side is larger than 0.38 and right side is smaller than 0.68, as represented by the gray area in Fig. 3. Note that the bottom-right corner cell is not included because a one-dimensional object's left bound cannot exceed its right bound. We map this area into a set of Z-value intervals, which is $\{[0, 16], [18, 18], [24, 24]$ $[26, 26], [32, 33], [36, 37]\}$. We can see that 36 is contained by one of the intervals while 20 is not. Therefore, this window query will retrieve object $[0.31, 0.44]$ but not object $[0.78, 0.9]$.

The main issue of dual-transformation is that it doubles the dimensionality of the data space, and data in high-dimensional space tend to have bad clustering properties and hence query performance is impaired.

2.3.3 *B^{dual}-tree.* The B^{dual}-tree [Yiu et al. 2008] also uses space-filling curves together with B^+-tree but has a different window query algorithm. Instead of first mapping the window query into one-dimensional space and then checking intersection in the mapped space, the B^{dual}-tree algorithm traverses the tree top-down and directly determines those nodes that intersect the window query. Specifically, the algorithm traverses the B^+-tree starting from the root and decomposes the key range corresponding to each accessed tree node into *"perfect blocks"*. Here, a "perfect block" is a hyper-square in the data space containing cells whose space-filling curve values are contiguous. Then, the perfect blocks are checked against the

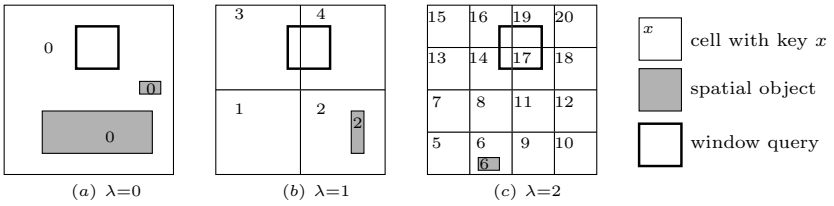


Fig. 5. SSSJ technique

window query, and the ones that interest the query window will have their corresponding child nodes in the tree accessed in the traversal. This process repeats and the objects intersecting the window query will be found when the traversal reaches the leaf nodes of the tree.

Fig. 4 shows an example where a B^+-tree node contains keys in the range of $[24, 49]$. The space corresponding to this key range $[24, 49]$ is divided into 5 perfect blocks whose curve values ranges are $[24, 27]$, $[28, 31]$, $[32, 47]$, $[48, 48]$, and $[49, 49]$, respectively. Since only blocks $[28, 31]$ and $[32, 47]$ intersect the window query, the query processing procedure only needs to further traverse the child nodes corresponding to these two blocks.

The intuition of the B^{dual}-tree query processing procedure is to avoid the expensive query window mapping operation by performing key range decomposition during B^+-tree traversal. However, the key range decomposition operation is still expensive, which has a time complexity of $O(2^{\mathcal{D}}\lambda)$, where $\mathcal{D}$ denotes the data dimensionality [Yiu et al. 2008]. Meanwhile, the number of times this decomposition operation is performed is proportional to the cardinality of the data set, which could be very large in real applications.

2.3.4 *Size Separation Spatial Join.* Size separation spatial join (SSSJ) [Koudas and Sevcik 1997] is an algorithm for computing the join between two spatial data sets. It can be applied to compute a window query by letting one of the data sets be of only one object, the window query. The key idea of SSSJ is to partition the objects roughly based on their size using grids of different granularity, and then index the objects on different grids with space-filling curves of different orders.

We use Fig. 5 to illustrate how SSSJ indexes objects using Z-curves. SSSJ has a predefined max Z-curve order, which is 2 in the example. Then SSSJ assumes Z-curves of orders 0, 1, ..., up to the predefined max Z-curve order 2, respectively, i.e., three conceptual grids are used to partition the data space into into 1, 4 and 16 cells respectively. Every data object is assigned to the grid with the smallest cell that can fully contain the object. As shown in the figure, the three grids are assigned with 2, 1, and 1 object denoted by the gray rectangles, respectively. SSSJ then maps the objects in different grids with the corresponding Z-curves, starting from the grid of order 0. The grid of order 0 has only 1 cell whose Z-value is 0. Thus, all objects in this grid are mapped to 0. The grid of order 1 has 4 cells. These 4 cells are numbered from 1 to 4 because Z-value 0 has been occupied by the previous grid. The object in this grid is in cell 2 and hence it is mapped to 2. Similarly the grid of order 2 has 16 cells whose Z-values are in the range of 5 to 20. The object in this grid is mapped to 6.

A window query is processed in SSSJ by mapping the window query to Z-value

intervals for each of the Z-curves used in the indexing and then continuing with the normal window query processing procedure of the Z-curve based indexing scheme.

SSSJ differs from our proposed indexing scheme SSI in several ways. First, a major drawback of SSSJ is that *actually, its separation strategy may not group objects strictly based on size.* As shown in Fig. 5a, two objects with quite different sizes are assigned to the same partition because they will both intersect more than one cells if assigned to any other partition. This violates the aim of size separation and may result in bad performance. In comparison, SSI strictly separates objects based on their size and thus avoids this problem. In SSI, we use the window query expansion approach [Stefanakis et al. 1997], which converts the problem of indexing objects with non-zero extents to indexing point objects. This allows us to index the centroid of each object, which lies in exactly one cell in any partition. Therefore, we avoid the above drawback of SSSJ and our separation is purely based on size. The query expansion paper [Stefanakis et al. 1997] also considered separating objects into multiple groups. However, their aim is to investigate how the window query should be expanded when the data objects have different sizes. In comparison, we use window query expansion and size separation to achieve appropriate orders of space-filling curves for indexing.

Second, the space-filling curves used in SSSJ may not handle skewed data well, which may cause many objects to be mapped to the same curve value. SSI employs a cost model to adaptively determine the best separation configuration, i.e., the number of separations and the size range allowed in each separation. Moreover, SSI maps skewed data to a space where data objects are distributed approximately evenly. This successfully addresses the problem of skewed data distribution.

2.3.5 *XZ-ordering.* XZ-ordering [Böhm et al. 1999] is another technique that puts objects of different sizes into grids of different granularity, and then indexes the objects on different grids with space-filling curves of different orders. To avoid assigning small objects intersecting the boundaries of small grid cells to grids with large cells, this technique extends the Z-curve by expanding every grid cell of the Z-curve by a factor of two in each dimension. As a result, adjacent cells of the same grid overlap with each other, and an object intersecting the boundary of one cell may be contained by one of the adjacent cells.

Compared with SSSJ, XZ-ordering groups objects based on size better. However, it may still have objects with quite different sizes to be assigned to the same partition. This is because an expanded grid cell may contain not only objects smaller than the original grid cells but also objects larger than the original grid cells. Thus, XZ-ordering is still *not* a strict separation based on object size. Also, XZ-ordering does not consider skewed data.

3. PRELIMINARIES

3.1 Problem Settings

We target managing spatial objects for real applications as described in Section 1, where objects have low dimensionality (typically two or three) and their extents cannot be ignored. Therefore, we focus on achieving high query performance for *2D and 3D objects with non-zero extents*, although our techniques work correctly in higher-dimensional spaces. We consider spatial objects that are relatively static,

i.e., we are not dealing with moving object databases, where frequent updates are the main issue [Koudas et al. 2004; Jensen et al. 2006; Nutanong et al. 2008]. Spatial objects with non-zero extents may have irregular shapes. We follow the common approach [Gaede and Günther 1998] to represent them by their *Minimum Bounding Rectangles (MBRs)*.

We focus on processing the spatial window query (the range query) since it is arguably the most basic and common spatial query and it could be used as a building block for other types of spatial queries such as the spatial join query and the (k) nearest neighbor query [Zhang et al. 2005]. We use the intersection predicate for the window query, i.e., we find all objects that intersect the window query. Applying our techniques to other types of predicates is straightforward.

Since we aim at a solution for full-fledged database systems, where databases are disk resident, our analysis and experimental study assumes disk-based algorithms. Nevertheless, our proposed algorithms are also applicable even if the data is fully held in main memory.

3.2 Drawbacks of Existing Space-filling Curve based Mapping Indexes

Our proposed Size Separation Indexing (SSI) scheme uses a space-filling curve to achieve a mapping index. We first analyze the drawbacks of existing space-filling curve based mapping indexes as follows.

(1) *Difficulty in choosing an order for the space-filling curve.* Space-filling curves with different orders have different granularity of object representation in the mapped space. A higher order may provide more accurate representation, but the computational cost of the mapping function is also higher. A lower order has lower mapping cost, but produces more false positives in a window query as objects with quite different positions may have the same mapped value. Achieving a balance between mapping efficiency and false positives, i.e., choosing the appropriate space-filling curve order, is tricky and requires enormous engineering effort in real applications.

(2) *Inefficiency on skewed data.* A space-filling curve uses a conceptual grid to partition the data space and then assigns each cell in the grid with a same value. If the spatial objects are skewed, either in the size or distribution, a space-filling curve alone may fail to map them to distinct one-dimensional values. Considering that the dense portion of a data set is more likely to be queried, failing to map objects in the dense portion properly will lead to low query processing efficiency.

(3) *Expensive query mapping.* A window query needs to be mapped to intervals of one-dimensional values to be processed upon the B^+-tree. A high cost method for this mapping may dominate the cost of query processing. Naively this mapping can be done by repeated calls to a mapping function that maps a grid cell to its curve value for all grid cells intersecting the window query. However, the high cost of repetitive mapping function calls makes the naive method prohibitive for online query processing. Several methods have been proposed [Ramsak et al. 2000; Lawder and King 2001] to improve the mapping efficiency, but their computational cost still grows rapidly with the order of space-filling curve used and are very expensive for a large data set consisting

Table I. Symbols used in SSI and their descriptions

Symbol	Description
$\mathcal{D}$	Data dimensionality
i	A partition number
j	A dimension number
$\mathcal{O}$	A spatial data set
$\|\mathcal{O}\|$	The cardinality of $\mathcal{O}$
o	A spatial object
$\|o\|$	The size of o
$o.c$	The centroid of o
$o.l_j, o.u_j$ $(j = 1, \cdots, \mathcal{D})$	The lower and upper bounds of o on dimension j
S_1, $\|S_1\|$	The sample set of $\mathcal{O}$ and its cardinality for determining the best separation configuration
f	A size separation configuration
$f.n$	The number of partitions of f
$f.\overrightarrow{d}$	The vector of separation size values of f
$\hat{f}$	The best separation configuration
$S_2, \|S_2\|$	The sample set and its cardinality for cumulative mapping function construction
$cdf_{i,j}()$	The CDF of partition i on dimension j
$\|\mathcal{Z}\|$	The size of the data space
v_i	The total number of grid cells for the first i partitions
$C()$ $(C^{-1}())$	A space-filling curve mapping (inverse mapping) function
$ssi(o)$	The index key of o
λ	The order of a space-filling curve
q	A window query
$q.l_j, q.u_j$ $(j = 1, \cdots, \mathcal{D})$	The lower and upper bounds of q on dimension j
n_q	The number of objects intersected by the window query
$\tilde{q}$	A typical window query
$\|\tilde{q}\|$	The size of $\tilde{q}$
$\|\|\tilde{q}\|\|$	The area/volume of $\tilde{q}$
p_{cur}	The scale of page access increase of a space-filling curve

of spatial objects in a large space.

In the next section, we will see how SSI addresses the above drawbacks.

4. SIZE SEPARATION INDEXING

We first provide an overview of SSI and then detail how SSI indexes objects and processes window queries in Sections 4.1 and 4.2, respectively. To avoid confusion we use i to denote a partition number and j to denote a dimension number. Other symbols frequently used in the elaboration are summarized in Table I.

The SSI indexing has three steps.

(1) *Size separation.* SSI separates the spatial objects into multiple partitions. It

first determines how many partitions the objects should be separated into and what are the size values that the objects should be separated at. A cost model is developed to help determining the size separation. Different from SSSJ, our object separation scheme guarantees that the objects are grouped **strictly** based on their sizes. Strictly grouping similar sized objects into one partition enables us to use a space-filling curve of a suitable order to index this partition. Then we can use space-filling curves of different orders to suit different partitions and hence address Drawback (1): difficulty in choosing one suitable order for the space-filling curve to index all the objects.

(2) *Cumulative mapping.* We transform the spatial objects in each partition into a space where they are uniformly positioned. A technique called cumulative mapping is used for this transformation. This avoids indexing skewed data and hence addresses Drawback (2): inefficiency on skewed data.

(3) *Space-filling curve mapping and object indexing.* After cumulative mapping, we further map objects of each partition to one-dimensional values using a space-filling curve of an appropriate order, and then index the resultant one-dimensional values from all partitions with a B^+-tree.

The SSI query processing has two steps.

(1) *Window query mapping.* We first map the window query to a set of key ranges in the B^+-tree constructed by SSI. The mapping should generate key ranges that cover the keys whose corresponding grid cells intersect the window query. We propose two novel mapping algorithms that reduce the number of grid cells that need to be checked for generating the key ranges. This addresses Drawback (3): expensive query mapping.

(2) *Range searches on the B^+-tree and filtering.* We perform range searches on the B^+-tree using the key ranges generated in window query mapping to find the candidate objects intersecting the window query. These candidate objects are checked against the window query directly to filter out false positives.

4.1 Indexing the Objects

In this subsection we detail the three steps of SSI indexing.

4.1.1 *Size separation.* Let $\mathcal{O}$ be a set of $\mathcal{D}$-dimensional objects. Given an object $o \in \mathcal{O}$, we define the *size* of o as the largest extent of o in all dimensions, denoted by $|o|$. Formally,

$$|o| = \max \{o.u_1 - o.l_1, o.u_2 - o.l_2, ... \, o.u_{\mathcal{D}} - o.l_{\mathcal{D}}\}.$$

Here, $o.u_j$ and $o.l_j$ denote the upper and lower bounds of o in the j^{th} dimension.

A *separation configuration* of $\mathcal{O}$, denoted by f, consists of the number of partitions $f.n$ and a vector of *separation size values* $f.\overrightarrow{d} = \langle f.d_1, ..., f.d_n \rangle$. A separation size value $f.d_i$ means that any object with a size smaller or equal to $f.d_i$ belongs to partition i. Let $pn(o)$ be a function that returns the partition number of o. Then,

$$pn(o) = i, f.d_{i-1} < |o| \leq f.d_i.$$

For example, in Fig. 6, the objects are separated into 3 partitions, i.e., $f.n = 3$. The maximum object sizes in these 3 partitions are 13, 28 and 55, respectively.

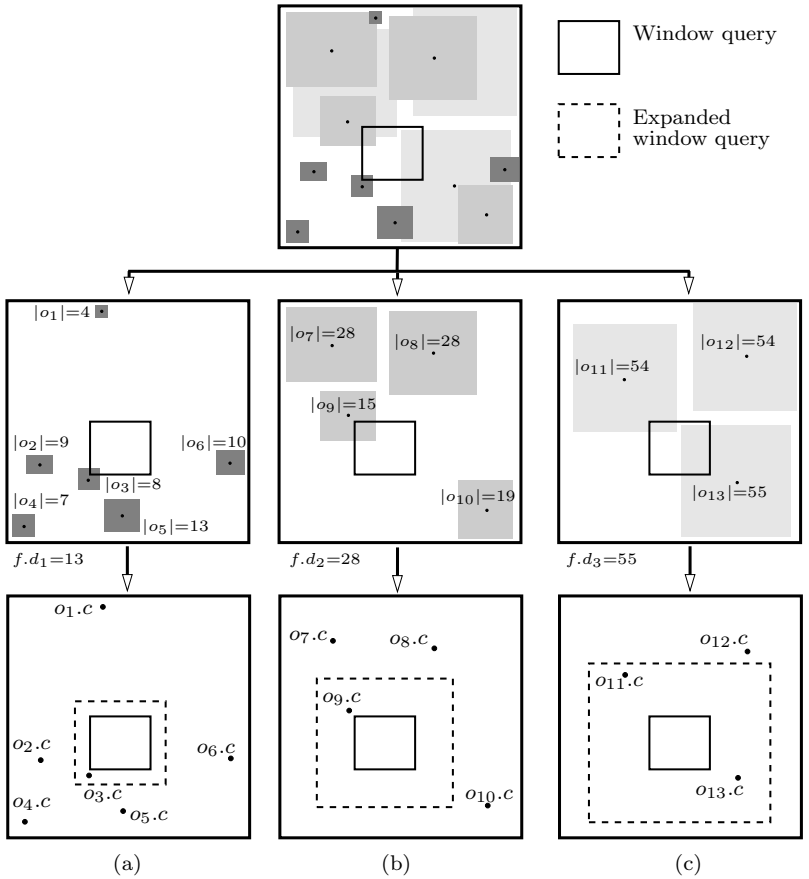


Fig. 6. A separation configuration ($f.n = 3$)

Therefore, the size values that the objects are separated at are 13, 28 and 55, i.e., $f.\overrightarrow{d} = \langle 13, 28, 55 \rangle$.

Different separation configurations may result in different query performance. To find the best separation configuration, SSI first uses sampling to obtain the size distribution of the objects in $\mathcal{O}$. Let S_1 be a random sample set on $\mathcal{O}$. Then we can estimate the size distribution of the objects in $\mathcal{O}$. If there are n_d objects in S_1 whose sizes are less than or equal to d, then there are approximately $n_d \times \frac{|\mathcal{O}|}{|S_1|}$ objects in $\mathcal{O}$ whose sizes are less than or equal to d, where $|\mathcal{O}|$ and $|S_1|$ denote the cardinality of $\mathcal{O}$ and $|S_1|$, respectively.

Then we use a two layer loop to exhaustively search for the best separation configuration. The outer loop iterates through different numbers of partitions (from 1 to n_{max}) while the inner for-loop iterates through different combinations of separation size values for each choice of number of separations. Here, n_{max} is a predefined system parameter indicating the largest possible number of partitions, while the separation size values are the sizes of the objects in S_1 obtained from the sampling step. In each iteration, a cost model (detailed in Section 5) is used to return the cost of the configuration given the number of partitions and the separation size values as input. After the two layer loop, the best separation configuration is found. As

the number of separation is usually small, typically 3 or 4 for many data sets and settings we have tested, and the number of separation size values is not too large either, typically below 500, this separation configuration selection process takes only tens of seconds to complete. Since it is done only once at indexing building time, the cost is acceptable.

Let $\hat{f}$ denote the best separation configuration found by the above algorithm. Then SSI separates $\mathcal{O}$ into $\hat{f}.n$ partitions, where the i^{th} partition contains the objects whose sizes are less than or equal to $\hat{f}.d_i$. At query time, a window expansion will be performed on each partition (detailed in Section 4.2). As a result of the expansion, we only need to check whether the centroids of the objects are in the expanded window query (as illustrated by the dash line rectangles in Fig. 6). This converts the problem of querying objects with non-zero extents to the problem of querying point objects. Therefore, the rest of our techniques only need to deal with points (the centroids of the original objects).

4.1.2 *Cumulative mapping.* The aim of this mapping is to achieve an approximately uniform distribution for the objects in a partition. SSI uses the Cumulative Distribution Function (CDF) for the mapping and thus the mapping is named *cumulative mapping*.

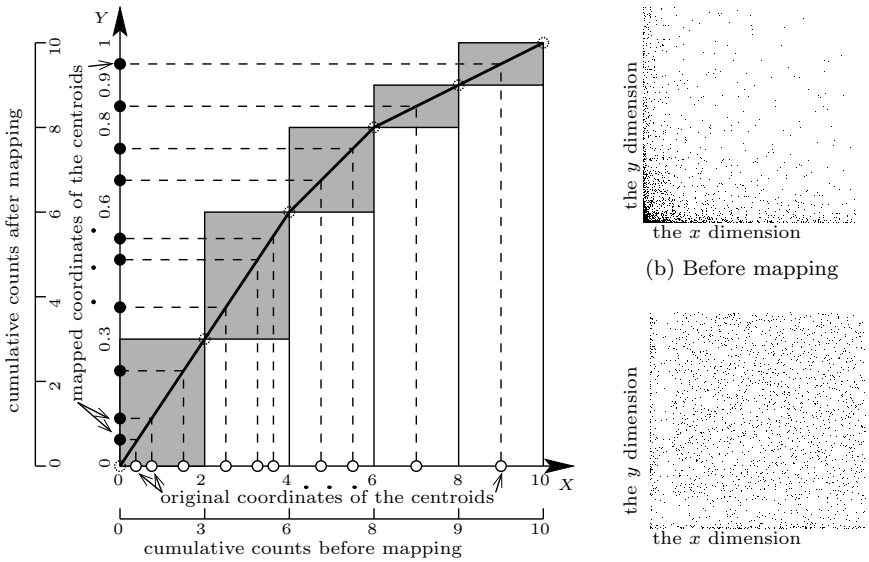
The cumulative distribution function $cdf(x)$ returns the percentage of data that are smaller than or equal to x. As discussed in Section 4.1.1, now we only deal with the centroids of objects. In each dimension j, we define a mapping $cdf_j(o)$, which returns the percentage of objects whose centroid coordinates are smaller than or equal to that of o in dimension j. Let $\langle o_1, o_2, ..., o_{|\mathcal{O}|}\rangle$ be a permutation of the objects in $\mathcal{O}$ in ascending order of their centroid coordinates in dimension j. Then,

$$cdf_j(o_k) = \frac{k}{|\mathcal{O}|}, k = 1, 2, ..., |\mathcal{O}|.$$

The CDF values of the objects are uniformly distributed in $[0, 1]$ in 1-dimensional space. Thus, we achieve a mapping that generates a uniform distribution. *Note that after the mapping, the whole data space is mapped into a unit hyper-cube space.*

Determining the CDF. Obtaining an exact CDF for cumulative mapping requires sorting all objects, which is expensive. Alternative, we only compute the CDF values at a small number of coordinate values, which can be obtained by a scan on the data set and hence avoids the sorting. Then, we use these CDF values to construct a *piecewise mapping function (PMF)* that approximates the exact CDF for the mapping.

We use an example as shown in Fig. 7a to illustrate how to construct the PMF. We evenly partition the data domain in dimension j into n_b buckets. Then the boundary coordinates of the buckets are the coordinates used to compute the PMF. In the figure, the data domain $[0, 10]$ in dimension X is divided into 5 buckets, and the boundary coordinates are $0, 2, 4, 6, 8$ and 10. To compute their CDF values, we first compute the cumulative count, which is the number of objects whose coordinates are smaller than or equal to a boundary coordinate. In the figure, the white dots denote the objects' centroid coordinates in dimension j. The cumulative counts of the boundary coordinates are $0, 3, 6, 8, 9$ and 10, respectively, which can be obtained by scanning the coordinates once. Dividing the total number of objects, 10, we get the CDF values of the boundary coordinates: $0, \frac{3}{10}, \frac{6}{10}, \frac{8}{10}, \frac{9}{10}, 1$. Then we plot the



(a) Mapping object centroids on a dimension (b) Before mapping (c) After mapping

Fig. 7. Cumulative mapping

CDF values of the boundary coordinates in dimension Y (denoted by the dashed dots), and connect the plotted points by line segments, which results in a polyline shown as the dark black polyline starting from the origin. This polyline corresponds to the PMF used to approximate the CDF for the mapping. As shown by the black dots on the Y axis, after mapping by this function, the objects are distributed approximately uniformly within the data domain $[0, 1]$.

Formally, the PMF to approximate the exact CDF is defined as follows. Let j be the current dimension for mapping, $|\mathcal{Z}|_j$ be the data domain size on the j^{th} dimension, $b_0, b_1, ..., b_{n_b}$ be the n_b+1 boundary coordinates of the buckets, and $b_k.c$ be the cumulative counts of b_k $(k = 0, ..., n_b)$. Then given an object o, its centroid $o.c$ is at $o.c_j$ in the j^{th} dimension, where $o.c_j = \frac{o.u_j + o.l_j}{2}$. Let $o.c$ be in bucket k.

$$k = \lfloor \frac{o.c_j}{|\mathcal{Z}|_j} \times n_b \rfloor. \tag{1}$$

The PMF used on the j^{th} dimension to map $o.c$ of an object o in partition i, denoted by $cdf_{i,j}(o.c)$, is formally defined as follows.

$$\begin{aligned} cdf_{i,j}(o.c) &= \frac{\frac{b_{k+1}.c - b_k.c}{b_{k+1} - b_k}(o.c_j - b_k) + b_k.c}{|\mathcal{O}_i|} \\ &= \frac{(b_{k+1}.c - b_k.c)(o.c_j - b_k) + b_k.c(b_{k+1} - b_k)}{|\mathcal{O}_i|(b_{k+1} - b_k)}. \end{aligned} \tag{2}$$

Here, $\mathcal{O}_i$ denotes the set of objects of partition i. Note that if $o.c_j = |\mathcal{Z}|_j$, then $k = n_b$ and hence b_{k+1} will be undefined. In this case, we do not use Equation (2) to compute $cdf_{i,j}(o.c)$ but directly define it to be 1.

To further reduce the cost of computing the PMF, we only computes the PMF

on a small sample set S_2. Then $cdf_{i,j}(o.c)$ becomes:

$$cdf_{i,j}(o.c) = \frac{(b_{k+1}.c - b_k.c)(o.c_j - b_k) + b_k.c(b_{k+1} - b_k)}{|S_2|(b_{k+1} - b_k)}. \quad (3)$$

As shown in Fig. 7b and Fig. 7c, after the cumulative mapping, a skewed 2-dimensional data set becomes an approximately uniform data set.

The approximation ratio of the PMF. We derive the *approximation ratio* of the PMF values over the CDF values, denoted by ρ, as follows. At the bucket boundaries, the PMF and the CDF have the same values. The approximation ratio ρ is 1. For a point between two bucket boundaries, ρ is derived as follows. Assume that the objects between two adjacent bucket boundary coordinates b_k and b_{k+1} are sorted in ascending order of their centroid coordinates. Let o be the m^{th} object and l be its distance to b_k. The CDF value of the centroid of o is $\frac{b_k.c + m}{|\mathcal{O}_i|}$. We can also compute the PMF value of the centroid of o by Equation (2). Then,

$$\rho = \frac{cdf_{i,j}(o.c)}{\frac{b_k.c + m}{|\mathcal{O}_i|}} = \frac{\frac{(b_{k+1}.c - b_k.c)(o.c_j - b_k) + b_k.c(b_{k+1} - b_k)}{|\mathcal{O}_i|(b_{k+1} - b_k)}}{\frac{b_k.c + m}{|\mathcal{O}_i|}}$$

$$= \frac{(b_{k+1}.c - b_k.c)l + b_k.c(b_{k+1} - b_k)}{(b_k.c + m)(b_{k+1} - b_k)}$$

From the equation above we can see that the exact value of ρ depends on the values of l and m of a specific object. We further derive upper and lower bounds of ρ that are irrelevant to any specific object. Since $1 \leq m \leq b_{k+1}.c - b_k.c$ and $0 < l < b_{k+1} - b_k$, we have:

$$\frac{(b_{k+1}.c - b_k.c)0 + b_k.c(b_{k+1} - b_k)}{(b_k.c + b_{k+1}.c - b_k.c)(b_{k+1} - b_k)} < \rho < \frac{(b_{k+1}.c - b_k.c)(b_{k+1} - b_k) + b_k.c(b_{k+1} - b_k)}{(b_k.c + 1)(b_{k+1} - b_k)}$$

$$\Rightarrow \frac{b_k.c}{b_{k+1}.c} < \rho < \frac{b_{k+1}.c}{b_k.c + 1}.$$

Therefore, the number of buckets and the number of objects within a bucket define two bounds of ρ, and we can use these bounds to help determine the parameter values when construing the PMF to achieve a certain value of ρ.

4.1.3 *Space-filling curve based mapping and object indexing.* After cumulative mapping, we further map the objects of each partition with a space-filling curve to obtain the index keys. The input of this mapping is the coordinates of an object o after cumulative mapping, and the output is the space-filling curve value of o. The curve values from different partitions are separated by adding the total number of grid cells in the first $(i-1)^{th}$ partitions to the curve values of partition i. This way, we only need one B$^+$-tree to index the objects from all partitions.

The challenge of this part is determining the order of the space-filling curve for each partition. Our query processing algorithm involves two conceptual phases:

(i) identifying the *cells* intersected by the window query and from these cells,

(ii) identifying the *objects* intersected by the window query.

As discussed in Section 3, a space-filling curve with *larger cells* (smaller order) has fewer cells and will make phase (i) less expensive, but it also has more objects

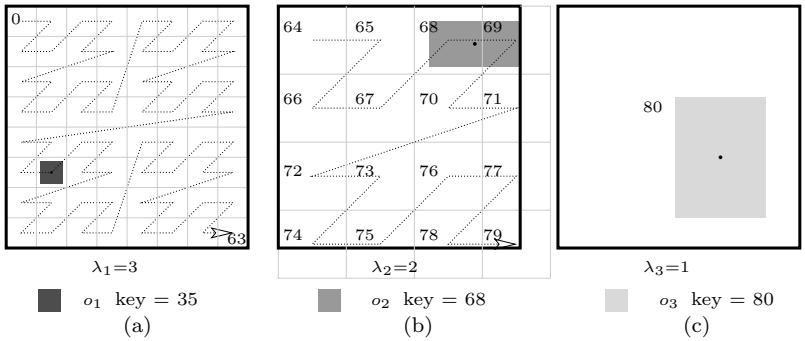


Fig. 8. Indexing three objects with different sizes using SSI ($\hat{f}.n = 3$)

in each cell and hence higher cost for phase (ii). A space-filling curve with *smaller cells* (higher order) has lower cost in phase (ii) but higher cost in phase (i).

To achieve a balance of the cost of the two steps, we propose to use the separation size value $\hat{f}.d_i$ as the cell size. The intuition is that we let a cell be large enough to enclose an object but not too large so as to avoid too many false positives in query processing. This way, every object will have only one index key. The size of the B$^+$-tree to store these keys is reduced and the query possessing efficiency is improved. Note that our indexing is on object centroids. Every object centroid is in one cell and every object has only one index key already. It seems that the cell size can be even below $\hat{f}.d_i$ to constrain false positives in query processing as much as possible. However, our query processing requires window query expansion. This effectively gives back the objects their extents. If the cells are too small, the expansion part of the window query will cover a lot of extra cells, which will result in high extra cost to map the window query (detailed in Section 4.2.1).

Next, we determine the order of the space-filling curve for partition i, denoted by λ_i. The conceptual grid corresponding to the space-filling curve should cover the whole data space. A space-filling curve of order λ_i has 2^{λ_i} cells on each side, and each cell has a side length of $\hat{f}.d_i$. Let $|\mathcal{Z}|$ be the data domain size. Then λ_i should satisfy $2^{\lambda_i}\hat{f}.d_i \geq |\mathcal{Z}|$, which means

$$\lambda_i = \lceil \log_2 \frac{|\mathcal{Z}|}{\hat{f}.d_i} \rceil.$$

As shown in Fig. 8b, a partition has a separation size value of 0.28 so the cell size is 0.28. To cover the data space, which has a side length of 1, the order of the space-filling curve should be $\lceil \log_2 \frac{|\mathcal{Z}|}{\hat{f}.d_i} \rceil = \lceil \log_2 \frac{1}{0.28} \rceil = 2$.

Note that the resultant conceptual grid of the space-filling curve may be a bit larger than the data space, which means a cell on the right or bottom side may only overlap with the data space with a smaller region. As a result, the cell may contain a smaller number of objects and hence cause an uneven index key distribution. However, the smaller overlapping region of the cell also means a smaller probability that this cell is intersected by the query window. Therefore, the impact of the extra space of the grid on query processing performance is limited.

We now have a space-filling curve with $(2^{\lambda_i})^{\mathcal{D}}$ cells. This is also the number of grid cells for partition i. Then, the total number of grid cells of the first i^{th}

partitions, denoted by v_i, is computed as:

$$v_i = \sum_{k=1}^{i} (2^{\lambda_i})^{\mathcal{D}} = \sum_{k=1}^{i} (2^{\lceil \log_2 \frac{|\mathcal{Z}|}{\hat{f}.d_i} \rceil})^{\mathcal{D}} \quad (4)$$

We add v_{i-1} to the curve values of the objects in partition i to avoid the overlap between the index key ranges of different partitions. As a result, after the space-filling curve mapping and the curve value adding, we have the index key of an object o of partition i, denoted by $ssi(o)$, computed as follows.

$$ssi(o) = C_i\big(\overrightarrow{cdf}_i(o.c)\big) + v_{i-1}. \quad (5)$$

Here, $\overrightarrow{cdf}_i(o.c) = \langle cdf_{i,1}(o.c), cdf_{i,2}(o.c), ..., cdf_{i,\mathcal{D}}(o.c)\rangle$ returns the centroid coordinates of an object o after cumulative mapping, and $C_i(\cdot)$ returns the space-filling curve value of a given point in the data space. Function $C_i(\cdot)$ depends on the type and order of the space-filling curve used for partition i. Once we have the index keys of all objects, we can put them into a B^+-tree to index the objects.

Fig. 8 shows an example where the number of partitions $\hat{f}.n$ is 3 and the orders of the space-filling curves (Z-curves in particular) used for the three partitions are 3, 2 and 1, respectively. After cumulative mapping, the centroid of object o_1 in partition 1 is at cell 35, i.e., $C_1(\overrightarrow{cdf}_1(o_1.c)) = 35$, in Fig. 8a. Since there is no previous partition, i.e., $v_0 = 0$, object o_1 is assigned the key 35 directly. Thus, $ssi(o_1) = 35$. The centroid of object o_2 in partition 2 is located in cell 4 in Fig. 8b, i.e., $C_2(\overrightarrow{cdf}_2(o_2.c)) = 4$, but partition 1 has occupied curve values 0 to 63, i.e., $v_1 = 64$. Therefore, we assign $ssi(o_2) = 4 + 64 = 68$. Similarly, we have $ssi(o_3) = 0 + 80 = 80$.

4.1.4 *The SSI Indexing Algorithm.* Algorithm 1 summarizes the size separation indexing process.

(1) *Size separation.* The algorithm starts with sampling a set S_1 from $\mathcal{O}$, which takes $O(|S_1|)$ time and space (lines 1 to 3). The size distribution of the objects in S_1 serves as an approximation of the size distribution of $\mathcal{O}$. Based on the approximated size distribution, a cost model is used to search for a best size separation configuration $\hat{f}$ by computing and comparing the expected cost of processing a window query under different configurations (lines 4 to 23). Here, for each possible number of partitions n, the different combinations of separation size values are generated using an algorithm that generates all size n combinations out of $|S_1|$ objects in the lexicographical order. The time complexity is $O(nC(|S_1|, n))$, where the $O(C(|S_1|, n))$ part denotes complexity of generating the combinations and the $O(n)$ part denotes the complexity of cost model computation for a combination. The space complexity is $O(n_{max})$, which is for the auxiliary array *chosen*[] used for combination generation. To further reduce the computational cost of generating the combinations, we may first build a histogram on the object sizes and then generate the separation size combinations on the histogram, so that the number of combinations is reduced. After the best separation configuration $\hat{f}$ has been identified, the data set $\mathcal{O}$ is separated into $\hat{f}.n$ partitions according to $\hat{f}.\overrightarrow{d}$, which takes $O(|\mathcal{O}|)$ time and space (line 24).

(2) *Cumulative mapping.* Objects of each partition are mapped first by cu-

Algorithm 1: Size Separation Indexing

Data: $\mathcal{D}$-dimensional object set $\mathcal{O}$,
Result: A B$^+$-tree indexing all object in $\mathcal{O}$

```
1  for k ← 0 to |S1| − 1 do                                   /* Sample object sizes */
2      randomly choose an object o from O
3      d[k] ← |o|
4  E* ← ∞                                                    /* Estimated optimal cost */
5  for n ← 1 to n_max do                                     /* Search for optimal configuration */
6      for i ← 0 to n − 1 do                                 /* Generate the first combination */
7          chosen[i] ← i + 1
8      flag ← 1
9      while flag = 1 do
10         compute a window query cost estimation E for the combination of d[] marked
           by chosen[]
11         if E < E* then
12             E* ← E, update f̂                              /* Save optimal */
13         chosen[n − 1] + +                                 /* Generate the next combination */
14         if chosen[n − 1] ≤ |S1| then
15             continue
16         i ← n − 1
17         while chosen[i] ≥ |S1| − (n − 1 − i) do
18             if − − i < 0 then
19                 flag ← 0, break
20         chosen[i] + +
21         while i < n − 1 do
22             chosen[i + 1] = chosen[i] + 1
23             i + +
24 separate O into f̂.n partitions according to f̂.d⃗
25 for i ← 1 to f̂.n do                                       /* Map each partition */
26     for j ← 1 to D do                                     /* Map each dimension */
27         compute a CDF for cumulative mapping of partition i, dimension j
28         forall o ∈ partition i do                         /* Cumulative mapping */
29             perform cumulative mapping on dimension j
30     forall o ∈ partition i do                             /* Space-filling curve based mapping */
31         o.key ← ssi(o)
32         insert ssi(o) into a B+-tree                      /* Index object */
```

mulative mapping to achieve an approximately uniform distribution (lines 25 to 29). Here, sampling a set S_2 and computing an approximated CDF function takes $O(|S_2|)$ time and space. We do this for $\mathcal{D}$ dimensions, which takes a total of $O(\mathcal{D}|S_2|)$ time and space. The cumulative mapping of an object takes $O(\mathcal{D})$ time and $O(\mathcal{D}+n_b)$ space, where $O(n_b)$ denotes the space used for storing the boundary coordinates. We do the mapping for all objects, which takes $O(\mathcal{D}|\mathcal{O}|)$ time and $O(\mathcal{D}|\mathcal{O}|+n_b)$ space.

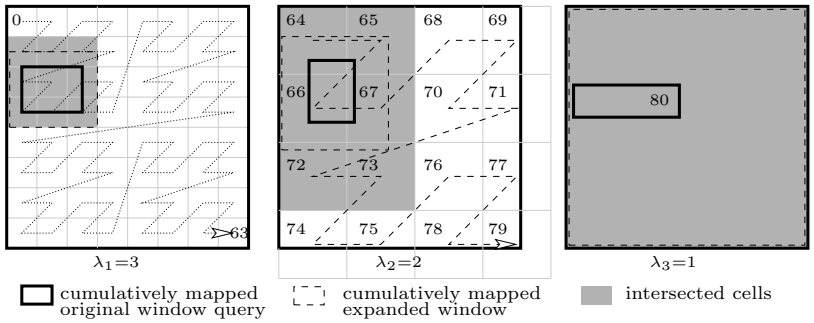


Fig. 9. Window query processing ($\hat{f}.n = 3$)

(3) *Space-filling curve mapping and object indexing.* The objects are then mapped by space-filling curve mapping (lines 30 and 31) to generate the index keys. The time and space complexities of this mapping is $O(|\mathcal{O}|time(C))$ and $O(|\mathcal{O}|space(C))$, where $O(time(C))$ and $O(space(C))$ denotes the time and space complexities of computing the curve value mapping function $C(\cdot)$, respectively. The generated keys are fed into a B$^+$-tree to index the objects in $\mathcal{O}$ (line 32), which takes $O(|\mathcal{O}| \log |\mathcal{O}|)$ time and $O(|\mathcal{O}|)$ space.

4.2 Processing Window Queries

A window query is first mapped to a set of key ranges in one-dimensional space. We then perform range search on the B$^+$-tree constructed by SSI to find all the objects intersecting the window query. Our focus here is how to map the window query to key ranges. The mapping should generate key ranges that cover all the grid cells intersecting the window query. As the grid cells for different partitions have different sizes, we map the window query for each partition based on its grid cell size. In the following discussion, we describe the mapping algorithm for one partition and apply the algorithm on every partition to generate all key ranges corresponding to the query.

First, we perform window query expansion to guarantee that we do not miss out objects whose centroids lie outside the window query q but actually intersecting q due to their extents. Specifically, given the separation size value $\hat{f}.d_i$ of partition i, we expand the lower and upper bounds of q on each dimension by $\frac{1}{2}\hat{f}.d_i$ as follows.

$$q.l_j \leftarrow q.l_j - \frac{1}{2}\hat{f}.d_i, \quad q.u_j \leftarrow q.u_j + \frac{1}{2}\hat{f}.d_i, \quad j = 1, 2, ..., \mathcal{D}.$$

Here, $q.l_j$ and $q.u_j$ denote the lower and upper bounds of q on dimension j, respectively. This achieves our goal as the distance between the centroid of an object and any of its bounds is less than or equal to half of the separation size value $\frac{1}{2}\hat{f}.d_i$.

Second, we perform the cumulative mapping on the sides of the window query using the CDFs constructed in the indexing stage. The monotonically increasing property of the constructed CDFs guarantees that no false negatives will be incurred by the cumulative mapping.

Third, the cumulatively mapped window query needs to be further mapped to key ranges on the B$^+$-tree based on the space-filling curve of partition i using a $MapRange$ function. Section 4.2.1 presents our proposed $MapRange$ algorithms.

Algorithm 2: SSI Window Query Processing

Data: A window query q, best separation configuration $\hat{f}$
Result: A Set $\mathcal{R}'$ of objects intersecting the window query

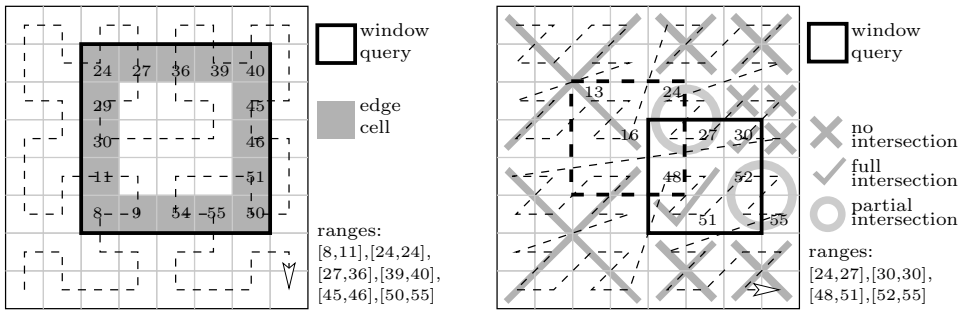
$\mathcal{K} \leftarrow \emptyset$
for $i \leftarrow 1$ **to** $\hat{f}.n$ **do**
  for $j \leftarrow 1$ **to** $\mathcal{D}$ **do** /* Cumulative mapping on query q */
    $qm.l \leftarrow q.l_j - \frac{1}{2}\hat{f}.d_i$
    $qm.u \leftarrow q.u_j + \frac{1}{2}\hat{f}.d_i$
    perform cumulative mapping on $qm.l$ and $qm.u$
  $\mathcal{K} \leftarrow \mathcal{K} \cup MapRange(qm, C_i())$ /* Get the index key ranges */
$\mathcal{R} \leftarrow$ perform range searches on B$^+$-tree with $\mathcal{K}$
forall $o \in \mathcal{R}$ **do** /* Filter false positives */
  if $o \cap q \neq \emptyset$ **then**
    $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{o\}$
return $\mathcal{R}'$

Fig. 9 gives an example for window query processing in a 2-dimensional space where there are three partitions on the data set. In each partition, every side of the window query is first expanded by $\frac{1}{2}\hat{f}.d_i$ to ensure no false negatives. Each side of the expanded window query is then mapped using the CDFs as defined by Equation (3). The *mapped expanded window query* (the dashed rectangle in Fig. 9) is then used to identify the cells in the space that intersect the window and generate the corresponding curve values. Note that the curve values are generated in the form of key ranges rather than a set of individual key values to reduce the number of searches performed on the B$^+$-tree. For example, the key ranges for partition 2 in the figure are [64, 67] [72, 73].

The key ranges obtained from window query mapping are then used to query the B$^+$-tree. The returned objects are compared with the window query in the original space, during which false positives are filtered and the final query result is found. Algorithm 2 summarizes the window query processing procedure, where qm denotes the mapped expanded window query.

Complexity. Expanding and cumulative mapping the window query takes $O(\hat{f}.n\mathcal{D})$ time and $O(\mathcal{D} + n_b)$ space, where the $O(\mathcal{D})$ part denotes the space used for the mapped expanded window query and the $O(n_b)$ part denotes the space used for the bucket boundary coordinates and their cumulative counts. Let $O(time(MapRange))$ be the time complexity and $O(space(MapRange))$ be the space complexity of the $MapRange$ function. Then mapping the mapped expanded window query to a set of 1-dimensional intervals $\mathcal{K}$ in the $\hat{f}.n$ partitions takes $O(\hat{f}.n \cdot time(MapRange))$ time and $O(space(MapRange))$ space. Querying the $|\mathcal{K}|$ intervals on the B$^+$-tree takes $O(|\mathcal{K}| \log |\mathcal{O}| + |\mathcal{R}|)$ time and $O(|\mathcal{O}|)$ space, where $\mathcal{R}$ denotes the set of objects to be returned from the B$^+$-tree queries. Checking the objects in $\mathcal{R}$ against the original window query takes $O(|\mathcal{R}|\mathcal{D})$ time and $O(|\mathcal{R}|\mathcal{D})$ space.

4.2.1 *MapRange Algorithms.* A straightforward method for window query mapping is as follows. Given a window query, we can identify all the intersected cells



(a) EdgeMapRange (b) RoughMapRange

Fig. 10. Example of the proposed *MapRange* algorithms

since we can easily compute the coordinates of the cell boundaries. For every intersected cell, we call the curve value mapping function $C()$ to obtain the curve value. All these curve values are then put together with contiguous ones combined into intervals. We call this method ***ScanMapRange***. The curve value mapping function $C()$ usually comes with an inverse mapping function $C^{-1}()$ that maps a curve value back to its corresponding cell in the curve. Examples of $C()$ and $C^{-1}()$ can be found in [Butz 1971; Faloutsos and Roseman 1989; Ramsak et al. 2000].

In what follows we propose two new *MapRange* algorithms. Since a desideratum of our index scheme is easy implementation on top of an existing DBMS, an important requirement for the *MapRange* algorithm is that the algorithm can *map a window query to a set of curve value intervals without accessing the data pages.*

1) ***EdgeMapRange***. We propose a *MapRange* algorithm called *EdgeMapRange* to improve the window query mapping efficiency for the Hilbert-curve. To the best of our knowledge, *this is the first MapRange algorithm using the Hilbert-curve without accessing data pages.*

This algorithm is based on the observation that, in a space-filling curve like the Hilbert-curve, the cells of consecutive curve values must be adjacent in space. We may infer the key ranges of the window query based on the cells on the edge (surface if in a data space of more than 2 dimensions) of the window query. An example is shown in Fig. 10a. Any two cells of consecutive curve values (e.g., cells 8 and 9, 54 and 55, etc.) are adjacent to each other. The window query contains six key ranges: $[8, 11], [24, 24], [27, 36], [39, 40], [45, 46]$, and $[50, 55]$. The boundary values of these ranges are all on the edge of the window query. To derive the key ranges, we first get the curve value for each cell on the window query edge using $C()$. For each cell c on the edge, we check whether it starts or ends a key range as follows. By calling $C^{-1}(C(c)-1)$ and $C^{-1}(C(c)+1)$ respectively we get the two cells preceding and following c in the curve. If the cell preceding c is not inside the window query, then the curve value of c starts a key range. If the cell following c is not inside the window query, then the curve value of c ends a key range. After checking every cell on the edge of the window query with the above procedure, we will have all key ranges. For example, in Fig. 10a, we can find that $8, 24, 27, 39, 45$, and 50 are the key range starting values, while $11, 24, 36, 40, 46$, and 55 are the key range ending values. Matching them up will give the key ranges of the window query.

Complexity. For a simple complexity analysis, assume that the window query is

a hyper-square and let n_c be the number of cells on a side of the window query. Then there are $n_c^{\mathcal{D}}$ cells in the window query, and the number of calls of $C()$ in $ScanMapRange$ is $O(n_c^{\mathcal{D}})$. The number of calls of $C()/C^{-1}()$ in $EdgeMapRange$ is $O(n_c^{\mathcal{D}-1})$. The improvement is most obvious when $\mathcal{D} = 2$, where the number of calls in $ScanMapRange$ is $O(n_c^2)$ while that in $EdgeMapRange$ is $O(n_c)$.

A limitation of $EdgeMapRange$ is that it only works for curves where the cells of consecutive curve values must be adjacent in space. For curves that do not satisfy this condition, the key range bounds may not be on the edge of the window query. For example, in the Z-curve in Fig. 10b, the window query represented by the dashed rectangle has a key range $[13, 16]$, but 16 is not on the edge of the window query. In this case, we seek for another $MapRange$ algorithm.

2) ***RoughMapRange***. We propose another $MapRange$ algorithm which is called *RoughMapRange*. This algorithm also does not require accessing the data pages, and it works for Z-curves.

The idea is that we map the window query to "rough" key ranges, which contains all the intersected cells but does not exactly cover them. Requiring only "rough" key ranges avoids computing lots of detailed key ranges while not introducing many false positives.

RoughMapRange recursively divides the data space into $2^{\mathcal{D}}$ sub-regions (e.g., for a 2-dimensional space, *RoughMapRange* divides a region into 4 sub-regions each time). Since this dividing strategy is the same as how a space-filling curve divides the data space, the key range corresponding to each sub-region is very simple to compute. For example, let a 2-dimensional data space be indexed by a space-filling curve of an order $\lambda = 3$. Then the key range of the data space is $[0, 2^{\lambda\mathcal{D}}-1] = [0, 63]$, while the key ranges of its four sub-regions are:

$$[0, \frac{2^{\lambda\mathcal{D}}}{2^{\mathcal{D}}} - 1], [\frac{2^{\lambda\mathcal{D}}}{2^{\mathcal{D}}}, 2\frac{2^{\lambda\mathcal{D}}}{2^{\mathcal{D}}} - 1], [2\frac{2^{\lambda\mathcal{D}}}{2^{\mathcal{D}}}, 3\frac{2^{\lambda\mathcal{D}}}{2^{\mathcal{D}}} - 1], [3\frac{2^{\lambda\mathcal{D}}}{2^{\mathcal{D}}}, 4\frac{2^{\lambda\mathcal{D}}}{2^{\mathcal{D}}} - 1],$$

which equal to $[0, 15]$, $[16, 31]$, $[32, 47]$, and $[48, 63]$, respectively.

For every region generated from the division, (i) if this region is fully covered by the window query, then the key range of this region is added to the output; (ii) if this region is not intersected by the window query, then this region is discarded; (iii) otherwise, the region is partly intersected by the window query. For case (iii), we use a cost model (details are in Section 5) to determine whether we should further divide the region or we should stop and simply add its key range to the output. This cost model estimates the cost of querying this region. If the cost estimated is not larger than one page access then we add the key range to the output. Otherwise, we divide the region further and check its sub-regions recursively using the previous steps. The above process ends when reaching the cell level of the data space.

Fig. 10b shows an example. The data space is divided because the window query does not fully cover the data space and the cost model estimates more than one page access on querying the whole data space. Next, we check the four sub-regions of the whole data space. Among them, the two on the left half of the data space are discarded because they do not intersect the window query. The other two sub-regions intersect the window query. They are subdivided again. The bottom-right sub-region contains (i) two sub-regions (on the bottom half) that do not intersect the window query and are discarded, (ii) one sub-region (on the top-left) that is fully covered by the window query and its key range is output, and (iii) one sub-region

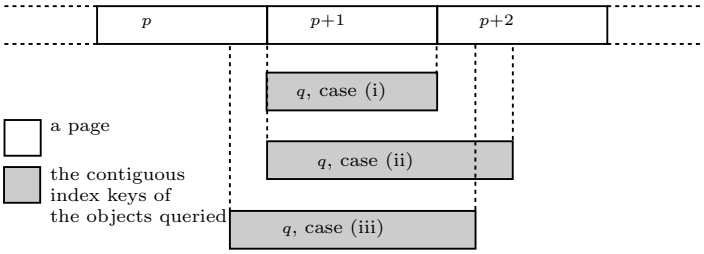


Fig. 11. Accessing objects intersected by the window query indexed contiguously in the B$^+$-tree

that is partly covered by the window query where the access cost is estimated to be less than one page and its key range is also output. The same procedure is applied to the rest of the data space and eventually, $\{[24, 27], [30, 30], [48, 51], [52, 55]\}$ is obtained as the *RoughMapRange* output.

RoughMapRange reduces the computational cost of mapping the window query by relaxing the key ranges corresponding to the window query. This might incur extra page accesses and introduce additional false positives. For example, in Fig. 10b, keys 24, 25, 53, and 55 are false positives. We will show in our experimental study that the performance gain of *RoughMapRange* outweighs the overhead of accessing the extra pages and filtering the additional false positives.

5. COST MODEL

In this section we present a cost model to estimate the number of index page accesses for processing a window query using SSI. This cost model is used for choosing the best parameters for SSI. It can also be used in the query plan evaluation if it is integrated in a DBMS. The cost model assumes that (i) the centroids of objects indexed in SSI follow an approximately uniform distribution, and (ii) given the same window query, the comparative performance of SSI in different separation configurations is mainly determine by the different space-filling curves used in those configurations. Assumption (i) is valid because we use cumulative mapping to transform the centroids of the objects to a space where they are approximately uniformly distributed. Assumption (ii) is valid because the query performance of a space-filling curve is fundamentally determined by an intrinsic property of the curve called the clustering property [Moon et al. 2001; Xu and Tirthapura 2012].

5.1 Overview of the Cost Model

The total number of page accesses of processing a window query, denoted by α, is the sum of the numbers of page accesses of processing the query on all partitions. We denote the number of page accesses of partition i by α_i.

Let n_q be the number of objects intersected by the window query q. Then α_i increases with n_q.

5.1.1 *Contiguous Cases.* Suppose that the n_q objects intersected by q are indexed contiguously in the B$^+$-tree and the capacity of a node (a page) in the tree is C_{max}. We derive the expectation of α_i, denoted by $E(\alpha_i)$, as follows.

As Fig. 11 shows, q may cover $\lfloor \frac{n_q}{C_{max}} \rfloor$, $\lfloor \frac{n_q}{C_{max}} \rfloor + 1$, or $\lfloor \frac{n_q}{C_{max}} \rfloor + 2$ nodes, depending on n_q and the starting position of the n_q objects in the nodes, denoted

by $s, s = 0, 1, ..., m, ...C_{max}-1$, Here, $s = m$ means that the object with the smallest key among the n_q objects is placed at the m^{th} entry of a node. We rewrite n_q to be $n_q = \lfloor \frac{n_q}{C_{max}} \rfloor C_{max} + k, k = 0, 1, ..., l, ...C_{max} - 1$. Then,

$$E(\alpha_i) = \sum_{l=0}^{C_{max}-1} \sum_{m=0}^{C_{max}-1} p(k = l, s = m)\alpha_{i,l,m},$$

where $p(k = l, s = m)$ denotes the probability of $k = l$ and $s = m$, and $\alpha_{i,l,m}$ denotes the number of leaf nodes accessed for the query on partition i when $k = l$ and $s = m$.

For a random window query, $p(k, s)$ should be the same for any combination of k and s in the Cartesian space of $[0, C_{max} - 1] \times [0, C_{max} - 1]$, i.e., $p(k = 0, s = 0) = p(k = 0, s = 1) = ... = p(k = C_{max} - 1, s = C_{max} - 1) = \frac{1}{C_{max}^2}$. Thus,

$$E(\alpha_i) = \frac{1}{C_{max}^2} \sum_{l=0}^{C_{max}-1} \sum_{m=0}^{C_{max}-1} \alpha_{i,l,m}.$$

Next, we derive $\sum_{m=0}^{C_{max}-1} \alpha_{i,l,m}$ for each value of l, and sum the resultant values up to obtain the value of $E(\alpha_i)$.

(i) When $l = 0$, if $m = 0$, then $\alpha_{i,l,m} = \lfloor \frac{n_q}{C_{max}} \rfloor$; otherwise, $\alpha_{i,l,m} = \lfloor \frac{n_q}{C_{max}} \rfloor + 1$.

$$\sum_{m=0}^{C_{max}-1} \alpha_{i,l,m} = \lfloor \frac{n_q}{C_{max}} \rfloor + (C_{max} - 1)(\lfloor \frac{n_q}{C_{max}} \rfloor + 1).$$

(ii) When $l = 1$, $\alpha_{i,l,m} = \lfloor \frac{n_q}{C_{max}} \rfloor + 1$, regardless of the value of m.

$$\sum_{m=0}^{C_{max}-1} \alpha_{i,l,m} = 0(\lfloor \frac{n_q}{C_{max}} \rfloor + 2) + C_{max}(\lfloor \frac{n_q}{C_{max}} \rfloor + 1).$$

(iii) When $l = 2$, if $m = C_{max} - 1$, then $\alpha_{i,l,m} = \lfloor \frac{n_q}{C_{max}} \rfloor + 2$ (i.e., the first and the last two objects each resides in one node, while the rest of the objects reside in $\lfloor \frac{n_q}{C_{max}} \rfloor$ nodes); otherwise, $\alpha_{i,l,m} = \lfloor \frac{n_q}{C_{max}} \rfloor + 1$.

$$\sum_{m=0}^{C_{max}-1} \alpha_{i,l,m} = (\lfloor \frac{n_q}{C_{max}} \rfloor + 2) + (C_{max} - 1)(\lfloor \frac{n_q}{C_{max}} \rfloor + 1).$$

(iv) Similarly, when $l > 2$, if $m \geq C_{max} - l + 1$, then $\alpha_{i,l,m} = \lfloor \frac{n_q}{C_{max}} \rfloor + 2$; otherwise, $\alpha_{i,l,m} = \lfloor \frac{n_q}{C_{max}} \rfloor + 1$.

$$\sum_{m=0}^{C_{max}-1} \alpha_{i,l,m} = (l - 1)(\lfloor \frac{n_q}{C_{max}} \rfloor + 2) + (C_{max} - l + 1)(\lfloor \frac{n_q}{C_{max}} \rfloor + 1).$$

We sum up the values of $\alpha_{i,l,m}$ and obtain:

$$\begin{aligned}\sum_{l=0}^{C_{max}-1}\sum_{m=0}^{C_{max}-1}\alpha_{i,l,m} &= \lfloor\frac{n_q}{C_{max}}\rfloor + (C_{max}-1)(\lfloor\frac{n_q}{C_{max}}\rfloor+1)\\ &+ \sum_{l=1}^{C_{max}-1}[(l-1)(\lfloor\frac{n_q}{C_{max}}\rfloor+2) + (C_{max}-l+1)(\lfloor\frac{n_q}{C_{max}}\rfloor+1)]\\ &= C_{max}^2\lfloor\frac{n_q}{C_{max}}\rfloor + \frac{3C_{max}^2}{2}(1-\frac{1}{C_{max}}).\end{aligned}$$

Thus,

$$E(\alpha_i) = \lfloor\frac{n_q}{C_{max}}\rfloor + \frac{3}{2}(1-\frac{1}{C_{max}}).$$

Therefore, on average, the number of the leaf nodes accessed is $\lfloor\frac{n_q}{C_{max}}\rfloor + \frac{3}{2}(1-\frac{1}{C_{max}})$. Since in a B$^+$-tree the number of the leaf nodes is much larger than that of the non-leaf nodes, and C_{max} is usually quite large, we have

$$\alpha_i \approx \lfloor\frac{n_q}{C_{max}}\rfloor + \frac{3}{2}. \tag{6}$$

5.1.2 *General Cases.* More generally, the objects intersected by q are not indexed contiguously in the B$^+$-tree because objects in the window query do not have strictly contiguous curve values. For example, in Fig. 10, a window query contains objects whose curve values are in multiple ranges. In this case the number of nodes to index these objects is larger than the number estimated by Equation (6). Thus, α_i increases. The scale of this increase, denoted by p_{cur}, is determined by how many of the objects intersected by the window query have contiguous curve values and how many do not, which in turn is determined by the *clustering property* of the space-filling curve *cur* used for object mapping. Studies on the clustering properties of different types of space-filling curves [Moon et al. 2001; Xu and Tirthapura 2012] have shown that different types of curves have different degree of contiguity on the curve values of the objects in a window query. Deriving a close form formula to compute the degree of contiguity of each space-filling curve and to accurately predict its impact on the scale of page access increase will be very difficult if possible at all. However, since the scale of page access increase is an inherent property of a particular space-filling curve which is a constant. We obtain the value of p_{cur} empirically.

Next we analyze how α_i changes with n_q and C_{max} in order to formulate α_i as a function of n_q, C_{max} and p_{cur}. Assume that the data objects are uniformly distributed in the data space, and that these objects are uniformly distributed in the leaf nodes of a B$^+$-tree. Then given a window query q, the number of leaf nodes that contain objects intersecting q, denoted by A, is proportional to the volume of q in a $\mathcal{D}$-dimensional space (area if $\mathcal{D}=2$), denoted by $v(q)$. Thus, we have $A \propto v(q)$. Meanwhile, the number of objects intersecting the window query, n_q, is also proportional to $v(q)$, i.e., $n_q \propto v(q)$. Therefore, we have $A \propto n_q$.

As the objects are uniformly distributed in the tree nodes, A is inversely proportional to the capacity of tree nodes, i.e., $A \propto \frac{1}{C_{max}}$. Together with $A \propto n_q$, we

have $A \propto \frac{n}{C_{max}}$.

Processing window query q requires accessing the leaf nodes that contain the objects intersecting q and some non-leaf nodes. Since in a B$^+$-tree the number of the leaf nodes is much larger than that of the non-leaf nodes, the number of *tree nodes* accessed for processing q, α_i, is approximately the number of *leaf nodes* accessed for processing q, i.e., $\alpha_i \approx A$. Thus, we have $\alpha_i \propto \frac{n_q}{C_{max}}$. From the analysis above we have also established that the scale of page access increase in the general case, p_{cur}, is a constant. Therefore, we have $\alpha_i \propto p_{cur}\frac{n_q}{C_{max}}$.

As a result, we can generalize Equation (6) and obtain

$$\alpha_i \approx p_{cur}(\lfloor \frac{n_q}{C_{max}} \rfloor + \frac{3}{2}). \tag{7}$$

5.2 Determining the Parameters in the Cost Model

We first derive the number of objects intersected by q, n_q, and then derive the scale of page access increase in the general case p_{cur}.

5.2.1 *The Number of Objects Intersected by the Window Query (n_q).* We derive n_q based on the ratio of the data space that is overlapped by the window query q. The idea is that, in a space where the data objects are uniformly distributed, the ratio of the objects contained in a window is approximately the ratio of the data space that is overlapped by the window [Tao et al. 2003]. In SSI, after cumulative mapping, the objects are approximately uniformly distributed. Therefore, $\frac{n_q}{n_i} \approx \frac{||q||}{||\mathcal{Z}||}$, where n_i, $||q||$, and $||\mathcal{Z}||$ denote the number of objects in partition i, the area (or volume) of the window query, and the area (or volume) of the data space, respectively. Since after cumulative mapping the area (or volume) of the data space $||\mathcal{Z}||$ becomes 1, we have:

$$n_q \approx n_i||q||. \tag{8}$$

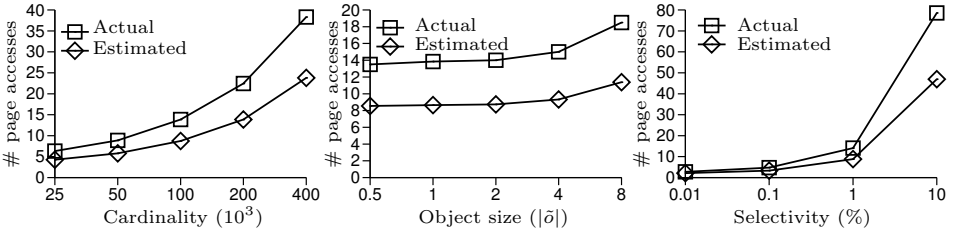
1) *Deriving* $||q||$. Based on the position of the lower bound $q.l_j$ and upper bound $q.u_j$ of q at every dimension j, we can perform window query expansion and cumulative mapping on q as described in Section 4.2 to get $||q||$. Formally,

$$||q|| = \prod_{j=1}^{\mathcal{D}} (cdf_{i,j}(q.u_j + \frac{1}{2}d_i) - cdf_{i,j}(q.l_j - \frac{1}{2}d_i)). \tag{9}$$

Here, $cdf_i, \cdot$ denotes the set of CDFs used for cumulative mapping in partition i, while d_i denotes the separation size value of partition i.

When we use the cost model for determining the separation configuration, we do not have a particular window query q yet and hence do not have any particular position of $q.l_j$ or $q.u_j$. We need an integration on all positions of $q.l_j$ and $q.u_j$. Formally,

$$||q|| = \prod_{j=1}^{\mathcal{D}} \int_0^{z_j} \int_{q.l_j}^{z_j} \frac{cdf_{i,j}(q.u_j + \frac{1}{2}d_i) - cdf_{i,j}(q.l_j - \frac{1}{2}d_i)}{|\mathcal{Z}|_j} d(q.l_j)d(q.u_j). \tag{10}$$



(a) Varying cardinality (b) Varying object size (c) Varying selectivity

Fig. 12. Estimation of p_{cur} for a 2D Hilbert-curve

Here, $|\mathcal{Z}|_j$ denotes the data space extent on dimension j.

2) *Simplifying* $||q||$. Equation (10) gives an accurate size of q for separation configuration selection. However, it is an expensive equation to compute. Since we need to use the cost model frequently in configuration selection, we simplify the computation of $||q||$ as follows. Our cost model assumes that data objects are uniformly distributed. The different positions of q should have similar selectivity effect on different partitions of the data set; the integration and cumulative mapping on the position of q do not affect the comparative result of the costs between different partitions much. Therefore, we drop the integration and CDFs from Equation (10), and replace $q.l_j$ and $q.u_j$ with $|q|_j$ to denote the extent of q on dimension j. This results in

$$||q|| = \prod_{j=1}^{\mathcal{D}} (|q|_j + d_i). \tag{11}$$

We do not have a particular window query at separation configuration selection and hence do not have a value for $|q|$. We use the size of a "typical window query" $\tilde{q}$ of a hyper-square shape, denoted by $|\tilde{q}|$. The intuition is that, we just need a cost comparison on different separation configurations of SSI to help determine the best configuration. The same "typical window query" should have the same effect on computing the cost of different configurations, and thus does not change the *comparative* cost of different configurations. Therefore, $||q||$ is further simplified.

$$||q|| = (|\tilde{q}| + d_i)^{\mathcal{D}}. \tag{12}$$

This equation has low computation overhead. In our experimental study, the results show that the value of $|\tilde{q}|$ has very little impact on the configuration selection results (Section 7.1.1).

5.2.2 *The Scale of Page Access Increase in General Cases (p_{cur}).* To learn the value of p_{cur} of a certain type of space-filling curve we implement SSI with the curve and then perform window queries using the implemented SSI. We record the actual number of page accesses and compare it with the number estimated by the cost model of the contiguous case (Equation (6), $\alpha_i \approx \lfloor \frac{n_q}{C_{max}} \rfloor + \frac{3}{2}$, in particular). We observe how large is the scale of page access increase in the general case when compared with the contiguous case.

Fig. 12 shows the result where we vary the data set cardinality, object size and window query selectivity to learn the value of p_{cur} for the Hilbert-curve in

Table II. Values of p_{cur} for different space-filling curves

Curve	Dimensionality	p_{cur}	Maximum Error
Hilbert	2	1.5722	5.45%
Z	2	1.8817	8.24%
Hilbert	3	3.2014	6.65%
Z	3	3.5883	7.15%

2-dimensional space[2]. In the figure, the actual and the estimated numbers of page accesses are denoted by "Actual" and "Estimated", respectively. We observe that the actual number is always about 0.6 times larger than the estimated number. We compute the scale of increase at every data point and take the average as the value of p_{cur}, which results in $p_{cur} = 1.5722$. We use this value in the cost model for the general cases (Equation (7), $\alpha_i \approx p_{cur}(\lfloor \frac{n_q}{C_{max}} \rfloor + \frac{3}{2})$) and re-estimate the number of page accesses for different settings. We compare the re-estimated numbers with the actual numbers, and the result shows that the maximum error rate is only 5.45%.

Similarly we obtain the values of p_{cur} for the Z-curve and the Hilbert-curve in 2-dimensional space and 3-dimensional space, respectively. The values and their corresponding maximum cost model estimation errors are listed in Table II. The achieved small estimation error rates (i.e., within 8.24%) validate the use of p_{cur} and the cost model.

5.3 Summary

Now we can integrate n_q and p_{cur} into Equation (7) to obtain the final forms of the cost model.

For separation configuration selection we use the simplified version of the cost model, where n_q is computed by Equations (8) and (12):

$$\alpha_i \approx p_{cur}(\lfloor \frac{n_i(|\tilde{q}| + d_i)^{\mathcal{D}}}{C_{max}} \rfloor + \frac{3}{2}). \tag{13}$$

For window query cost estimation we use the full version of the cost model, where n_q is computed by Equations (8) and (9):

$$\alpha_i \approx p_{cur}(\lfloor \frac{n_i(\prod_{j=1}^{\mathcal{D}}(cdf_{i,j}(q.u_j + \frac{1}{2}d_i) - cdf_{i,j}(q.l_j - \frac{1}{2}d_i)))}{C_{max}} \rfloor + \frac{3}{2}). \tag{14}$$

6. DBMS IMPLEMENTATION

In this section we describe how SSI can be implemented in a full-fledged DBMS without modifying the DBMS kernel.

We first create a table to store the MBRs of the spatial objects, where the lower bounds and upper bounds of the MBRs on each dimension are stored in separate columns. Another column is needed to store unique `id`s for associating the MBRs

[2]In Fig. 12b, $|\tilde{o}|$ denotes the average object size obtained from some real data sets, which will be detailed in Section 7.

with their corresponding spatial objects. These are all columns needed for enabling spatial queries on a full-fledged DBMS with no spatial support. In addition, SSI requires a column `ssi_key` in the table to store the SSI keys of the MBRs. Below we give an example SQL statement to create a table that stores 2D objects for SSI.

```
CREATE TABLE table_name (id INTEGER  UNIQUE,
                         ssi_key INTEGER,
                         lower_1 DOUBLE,
                         upper_1 DOUBLE,
                         lower_2 DOUBLE,
                         upper_2 DOUBLE);
```

We then create a B^+-tree index on `ssi_key`.

```
CREATE INDEX index_name ON table_name(ssi_key);
```

We may create tables to store the parameters for the SSI key computation. As shown in Equation (5), the parameters include the number of partitions $\hat{f}.n$, the data dimensionality $\mathcal{D}$, the number of samples used for building the CDFs, $|S_2|$, the number of buckets used for building the CDFs, n_b, the data space size in each dimension $|\mathcal{Z}|_j$, the order of the space filling curve used for each partition λ_i, the total number of grid cells used in the first i partitions v_i, and the cumulative counts for each bucket boundary coordinate $b_k.c$. We give example SQL statements of creating tables to store these parameters as follows.

```
CREATE TABLE ssi_overall_parameter (f_n INTEGER, D INTEGER,
                S_2 INTEGER, n_b INTEGER);
CREATE TABLE ssi_data_space_size (Z_j DOUBLE);
CREATE TABLE ssi_curve_parameter (lambda_i INTEGER, v_i INTEGER);
CREATE TABLE ssi_cdf_parameter (b_k_c INTEGER);
```

Given a data set, we have a C program to (i) generate the parameters for SSI key computation, (ii) compute the SSI key for each record, and (iii) insert the parameters as well as the SSI keys into the database tables in a DBMS through the DBMS' programming interface such as ODBC.

The above program is also used for data maintenance. When a new data record arrives, the program computes the SSI key for the record and generates an `INSERT` statement to insert it into the database table; when a record is to be deleted, the record `id` is passed to the program and the program generates a `DELETE` statement to delete the record from the database table; when a record's MBR is updated, the program first computes a new SSI key for the record based on the new MBR, and then generates an `UPDATE` statement to update the record in the database table. We give example SQL statements for data maintenance as follows:

```
INSERT INTO table_name VALUES (o_id, o_ssi_key,
                o_lower_1, o_upper_1, o_lower_2, o_upper_2);
DELETE FROM table_name WHERE id = o_id;
UPDATE table_name SET ssi_key = n_ssi_key,
                lower_1 = n_lower_1, upper_1 = n_upper_1,
```

```
                lower_2 = n_lower_2, upper_2 = n_upper_2
WHERE id = o_id;
```

Here, `o_id`, `o_ssi_key`, `o_lower_1`, `o_upper_1`, `o_lower_2` and `o_upper_2` denote the object `id`, `ssi_key` and MBR boundaries, while `n_ssi_key`, `n_lower_1`, `n_upper_1`, `n_lower_2` and `n_upper_2` denote the new `ssi_key` and MBR boundaries of an object.

At query time, we have another C program to process the window queries. Given a window query `[w_lower_1, w_upper_1; w_lower_2, w_upper_2]`, this C program uses either *EdgeMapRange* or *RoughMapRange* to generate the corresponding SSI key ranges, and then constructs a SQL query statement that contains all generated key ranges to perform a window query on the database table through the DBMS' programming interface as follows:

```
SELECT  id
FROM    table_name
WHERE   (ssi_key BETWEEN i_lower_1 AND i_upper_1 OR
        ssi_key BETWEEN i_lower_2 AND i_upper_2 OR
        ...
        ssi_key BETWEEN i_lower_k AND i_upper_k)
        AND (lower_1 <= w_upper_1 AND
        upper_1 >= w_lower_1 AND
        lower_2 <= w_upper_2 AND
        upper_2 >= w_lower_2);
```

Here, `<i_lower_1, i_upper_1>, ..., <i_lower_k, i_upper_k>` represent the ranges of space-filling curve values that the window query is mapped to. The final `AND` clause is needed because the comparisons on `ssi_key` may return false positives, which need to be filtered by the window query directly.

Note that the above implementation of SSI is on top of an existing DMBS, which does not change the kernel of the DBMS. We discuss the transaction management issue as follows. With the above SSI implementation, a spatial database operation (i.e., insert, update, delete, and window query) is carried out in two steps: (i) our C program computes the SSI key for an object, or a set of key ranges in the case of a window query, and generates the corresponding SQL statements; (ii) the DBMS executes the generated SQL statements to update the index. The full ACID property of a transaction containing the above spatial operations can be achieved by applying transaction models such as 2PL and failure recovery mechanisms outside the DBMS, e.g., in the application layer where the DBMS is queried, or a service layer which wraps the access to the DBMS. If applied in the application layer, many platforms such as Spring for JavaEE can be used, which takes care of the transaction management in the business logic, and therefore little additional effort is needed. If applied in an independent service layer, tremendous effort can be avoided since the B^+-tree implementation in the DBMS is leveraged to provide the ACID property in the second step. Overall, for the above SSI implementation, the transaction management can be achieved with a small amount of effort outside the DBMS. Alternatively, the SSI index technique can be integrated into the DBMS kernel (instead of on top of the DBMS) and let the DBMS handle the transaction

management and failure recovery issues. This may offer even better efficiency of the SSI index technique.

Beyond supporting the ACID properties, SSI brings little negative impact to the transaction throughput to an RDBMS. Specifically, we discuss the logging and the locking issues as follows. As for locking when processing the window queries, both SSI and the mapping-based competitive methods, e.g., XZ-ordering, map the query window to a set of key values for searching on the B^+-tree, which may incur shared locks on the corresponding values when performing transactions. The cost of such locks depends on the number of key values generated by the different methods. Given the same query, a method that generates more key values may have larger overhead of locking, and hence smaller degree of concurrency. Given the same data distribution and query, different mapping based methods should generate the same number of key ranges as the number of continuous space-filling curve intervals inside a window query remains the same irrespective of the mapping methods. Note that SSI has a distribution transformation step which makes the data more uniform, so SSI tends to generate a smaller number of continuous space-filling curve intervals compared to other mapping methods. This is confirmed by the experimental results in Section 7.2. When processing the same query SSI typically involves less page accesses, i.e., less false positives. Hence, SSI is expected to perform no worse than the other mapping based methods in terms of the degree of concurrency and the overhead of locking. R-Tree based methods are not competitive to SSI in terms of the concurrency control because the R-Tree is inherently much more complicated than the B^+-tree. The overlaps between MBRs in an R-Tree may cause many nodes to be accessed concurrently and therefore locked during the query time.

7. EXPERIMENTAL STUDY

In this section we present the results of an experimental study on SSI. We first study the effect of the different components on the performance of SSI in Section 7.1. Then we compare the query processing efficiency of SSI with that of some existing spatial indexes when implemented as a standalone spatial index in Section 7.2. Finally, we integrate SSI into two off-the-shelf DBMSs and compare the query processing efficiency with that of the spatial component of the two DBMSs. For each experiment we run 200 window quires and report the average number of page accesses and response time.

Our experimental system has an Intel Core^{TM} 2 Duo E600 processor, 2GB RAM, and a 7200RPM SATA hard drive with a page of 4096 bytes. All the algorithms are implemented in C and compiled using GCC 4.2.3 under Linux 2.6.24. Each float typed variable occupies 4 bytes in main memory. By default the indexes do not use buffers (I/O buffering of the operating system is allowed). In the experiments where we study the effect of buffer size, we vary the buffer size from 0 to 1200 pages and use Direct I/O to bypass the I/O buffering of the operating system, i.e., we use system function `open()` to open data files and set the `O_DIRECT` flag.

We use three real data sets obtained from the R-tree Portal[3]: Germany hypsography data set, Tiger/Line LA rivers and railways data set, and Tiger streams data set. These data sets contain 76,999, 128,971 and 194,971 2-dimensional MBRs, and

[3]http://www.chorochronos.org/

Table III. Data sets and parameters used in experiments

Cardinality	25000, 50000, **100000**, 200000, 400000, 1000000, 10000000
Dimensionality	**2**, 3
Distribution	**uniform**, Zipfian, real
Real data sets	Hypsography, **Railway**, Stream
Selectivity	0.01%, 0.1%, **1%**, 10%
Space-filling curves	**Z-curve**, Hilbert-curve
θ_d	0.2, 0.4, 0.6, **0.8**
θ_a	0.2, 0.4, 0.6, **0.8**
n_{max}	4
n_b	$\min\{\|\mathcal{O}\|, 5\log_2\|\mathcal{O}\|\}$
$\|S_1\|$	$\min\{\|\mathcal{O}\|, 50\log_2\|\mathcal{O}\|\}$
$\|S_2\|$	$\min\{\|\mathcal{O}\|, 25\log_2\|\mathcal{O}\|\}$

are denoted by "Hypsography", "Railway" and "Stream" data sets, respectively. We generate 3-dimensional real data sets from the 2-dimensional real data sets as follows. For each object o in a real data set, we use the size of an object randomly chosen from the same data set as the extent of o in the 3^{rd} dimension. We then randomly place the object in the 3^{rd} dimension within the range of $[0, |\mathcal{Z}|]$, where $|\mathcal{Z}|$ denotes the data space size of the corresponding 2-dimensional real data set.

We also generate 2- and 3-dimensional synthetic data sets with uniform and Zipfian distributions, respectively. In the uniform data sets, the object coordinates and extents in each dimension follow the uniform distribution in the range of $[0, 1]$. In the Zipfian data sets, the object coordinates and extents in each dimension follow the Zipfian distribution in the range of $[0, 1]$. We vary the data set cardinality from 25,000 to 10,000,000 and the skewness parameter of the Zipfian data sets, denoted by θ_d, from 0.2 to 0.8. Further, to evaluate the effect of object aspect ratio distribution, we generate Zipfian data sets where the object aspect ratio also follows the Zipfian distribution and the skewness parameter θ_a varies from 0.2 to 0.8.

For window query performance study we generate window queries of selectivity varying from 0.01% to 10%.

We test both the Z-curve and the Hilbert-curve as the space-filling curve used in SSI. To constrain the index building and SSI key computation time, in the size separation stage, we set the maximum number of partitions n_{max} to be 4 and the sample set cardinality for size distribution estimation $|S_1|$ to be $\min\{|\mathcal{O}|, 50\log_2|\mathcal{O}|\}$; in the cumulative mapping stage, we set the number of buckets to be $\min\{|\mathcal{O}|, 5\log_2|\mathcal{O}|\}$ and the sample set cardinality $|S_2|$ to be $\min\{|\mathcal{O}|, 25\log_2|\mathcal{O}|\}$ for cumulative mapping function construction; in the cost model we set the typical window query size $|\tilde{q}|$ to be $\frac{1}{64}$ of the data space size.

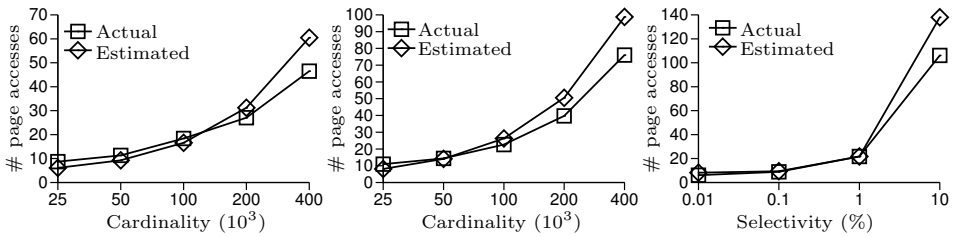
Table III summarizes the parameters used in the experiments, where the default values are in bold.

7.1 Effect of the Components of SSI

In this subsection we present the effect of three SSI components on query processing performance: cost model, cumulative mapping, and *MapRange* algorithms.

Table IV. Relative standard deviations when varying typical window size ($|\tilde{q}|$)

Test Case	Number of Page Accesses	Response Time
uniform 2D data set	0.00%	2.10%
Zipfian 2D data set	0.24%	1.72%
Railway 2D data set	1.06%	2.68%



(a) uniform 2D data set (b) Zipfian 2D data set (c) Railway 2D data set

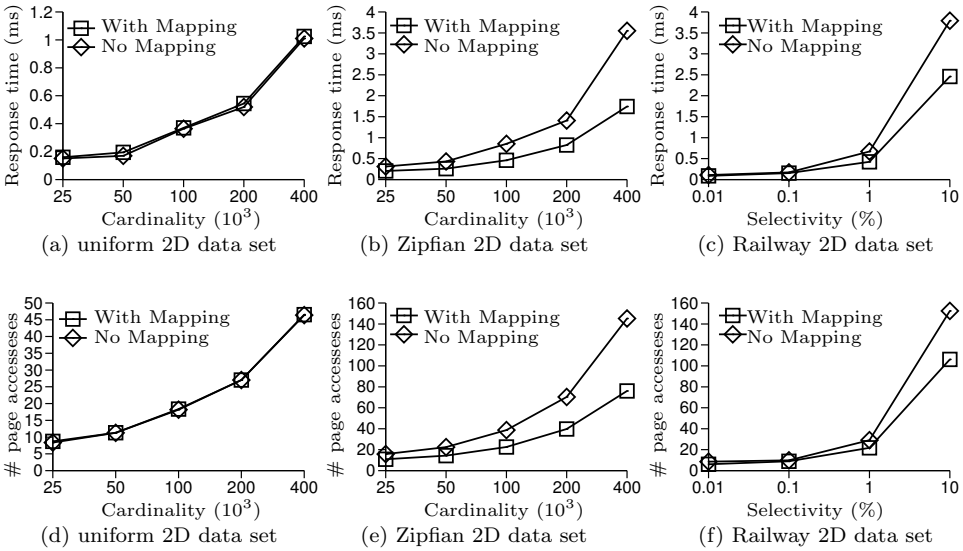
Fig. 13. Cost model accuracy

7.1.1 *Effect of the Cost Model.* **1) Typical window size.** In separation configuration selection, we used a typical window query size $|\tilde{q}|$ in the cost model. In this set of experiments we verify that the choice of a particular value of $|\tilde{q}|$ does not affect much on the result of the configuration selection. We vary the value of $|\tilde{q}|$ from $\frac{0.1}{64}$ to $\frac{10}{64}$ and observe the separation configuration selected by the cost model. We repeat the experiment on different data sets and find that, on each data set, the separation configuration selected is almost always the same. This is because although the estimated cost of a separation configuration varies when $|\tilde{q}|$ is varied, the *comparative* cost of different separation configurations stays the same and hence the selection result does not change.

For the cases where the selection result changes, we setup SSI with the different configuration selected and measure their query processing performance. Table IV shows the relative standard deviation of the number of page accesses and the response time of SSI of different configurations on 2-dimensional data sets (3-dimensional data sets give similar results and thus are omitted).

We can see that the different configurations selected result in little performance difference of SSI. The number of page accesses only varies less than 1.1%, while the response time only varies less than 3%. Therefore, even in rare cases where the choice of a particular value of $|\tilde{q}|$ may result in a sub-optimal separation configuration, it has very little impact on the performance of SSI.

2) Cost model accuracy. Next we evaluate the accuracy of our cost model on different data sets with different parameter settings. We record the numbers of actual page accesses incurred (denoted by "Actual"), and then compare them with the numbers of page accesses estimated by the cost model (denoted by "Estimated"). As Fig. 13 shows, the numbers of observed page accesses are very similar to their cost model estimated counterparts. The relative error on 2-dimensional data sets is within 30%. In particular, on uniform data sets, the error is the smallest, while on Zipfian and real data sets, the error is slightly larger. This is expected



(a) uniform 2D data set (b) Zipfian 2D data set (c) Railway 2D data set

(d) uniform 2D data set (e) Zipfian 2D data set (f) Railway 2D data set

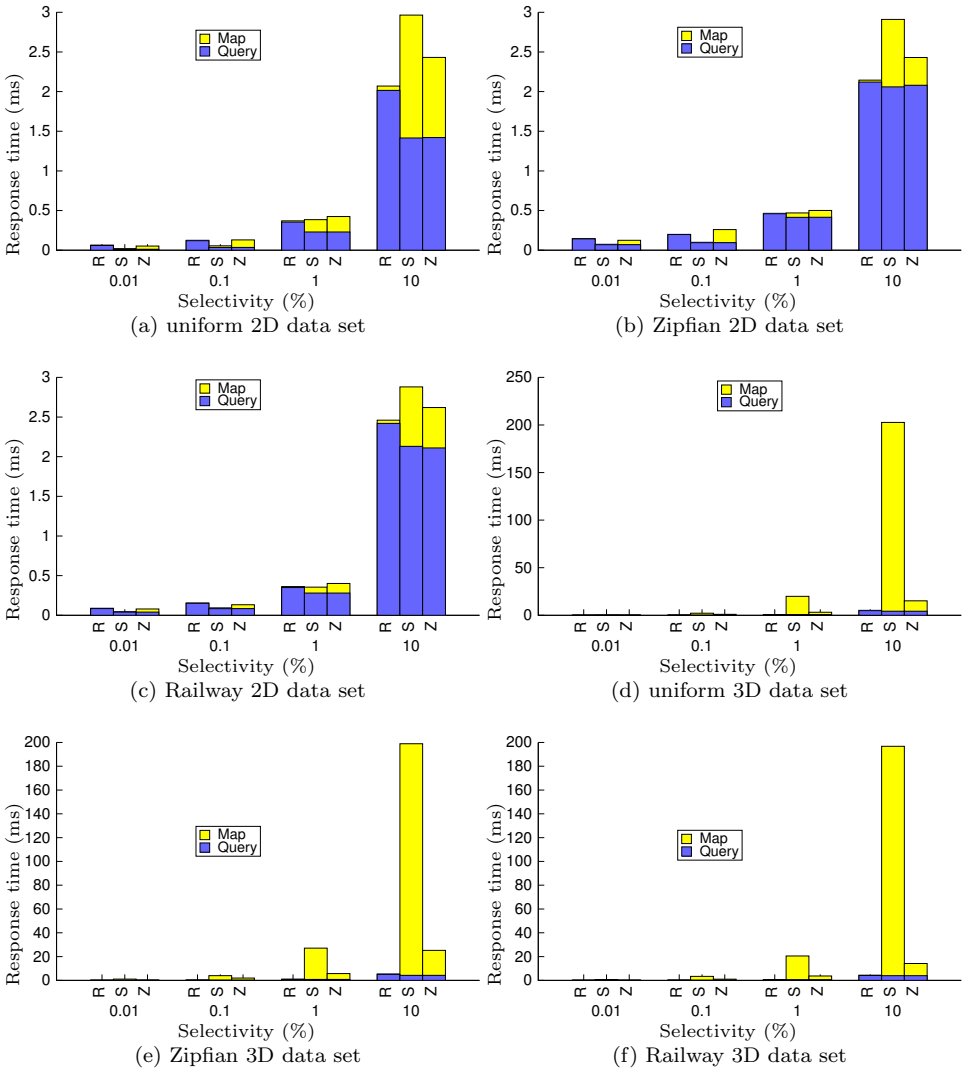
Fig. 14. Effect of cumulative mapping on window query performance of SSI

because the cost model is based on uniform data distribution. For non-uniform data sets, cumulative mapping can help obtain an approximately uniform distribution but it is not a strict uniform distribution. Thus, the accuracy of the cost model is slightly lowered. However, as the experiments in the following section will show, the achieved cost model accuracy is good enough to produce an index structure that significantly outperforms other spatial indexes. Experiments on 3-dimensional data sets show similar patterns and thus the result is omitted.

7.1.2 *Effect of Cumulative Mapping.* We evaluate the query processing performance of SSI with and without cumulative mapping to justify the use of cumulative mapping. As shown in Fig. 14, SSI with cumulative mapping outperforms SSI without cumulative mapping in most cases in terms of both the number of page accesses and the response time. For the data sets with Zipfian distribution and large cardinality, the effect of cumulative mapping is most obvious (up to 60% improvement in response time on the Zipfian data set with 400,000 objects, cf. Fig. 14b). This is because mapping based indexes do not perform very well on highly skewed data, but with cumulative mapping, we achieve an efficient mapping based index.

There is also considerable improvement on the real data set (37% improvement in response time when the query selectivity is 10%, cf. Fig. 14c) as real data are usually skew. For uniform data sets the effect of cumulative mapping is minimum. SSI with cumulative mapping is slightly slower because it needs cumulative mapping on the window query (computing Equation (3)), but the difference is barely observable (cf. Fig. 14a and Fig. 14d).

7.1.3 *Effect of MapRange Algorithms.* To evaluate the performance of different *MapRange* algorithms, we vary the query selectivity from 0.01% to 10%. We measure the response time and the number of page accesses of five *MapRange* algorithms in SSI to process window queries, i.e., *ScanMapRange*, *EdgeMapRange*,



(a) uniform 2D data set (b) Zipfian 2D data set

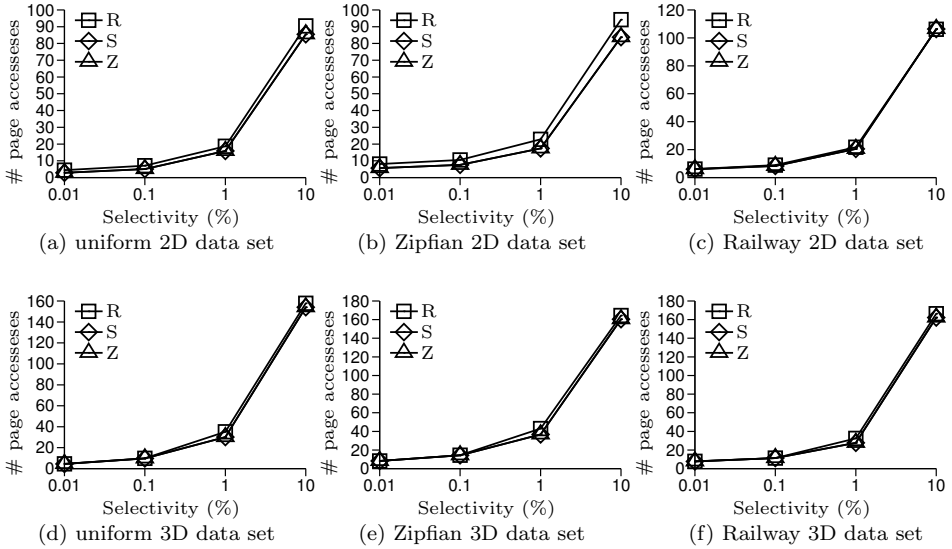
(c) Railway 2D data set (d) uniform 3D data set

(e) Zipfian 3D data set (f) Railway 3D data set

Fig. 15. Effect of *RoughMapRange* on window query performance of SSI - response time

RoughMapRange, *GetNextH* and *GetNextZ*. In the results we use "S", "E", "R", "H" and "Z" to denote the five algorithms, respectively. We show the time for window query mapping as well as B^+-tree searching, as denoted by "Map" and "Query", respectively. Here, *GetNextH* is a *MapRange* algorithm adapted from the *calculate_next_match* algorithm [Lawder and King 2001] for the Hilbert-curve and *GetNextZ* [Ramsak et al. 2000] is an existing *MapRange* algorithm for the Z-curve (cf. Section 2.3.1). Note that our study focuses on *MapRange* algorithms that do not require accessing the data pages. Therefore, existing algorithms that access the data pages during mapping are not considered.

1) First we compare *RoughMapRange* with *ScanMapRange* and *GetNextZ*. We report both the response time and the number of page accesses to evaluate how



(a) uniform 2D data set (b) Zipfian 2D data set (c) Railway 2D data set

(d) uniform 3D data set (e) Zipfian 3D data set (f) Railway 3D data set

Fig. 16. Effect of *RoughMapRange* on window query performance of SSI - page access

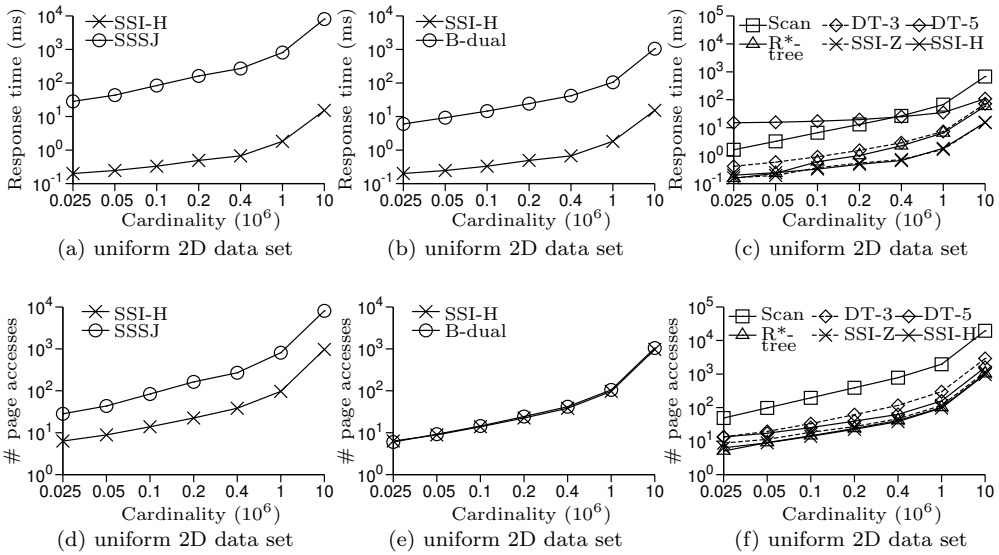
much performance gain in response time can be achieved by *RoughMapRange* and how much page access overhead it has.

Fig. 15 shows the response time. On 2-dimensional data, all three methods have similarly small response time when the query selectivity is less than 1%. When the query selectivity gets larger, the response time of *ScanMapRange* increases the fastest while that of *RoughMapRange* increases the slowest. That of *GetNextZ* lies in between the two. On 3-dimensional data, *RoughMapRange* shows clearer advantage. It outperforms *ScanMapRange* by an order of magnitude and *GetNextZ* by more than 3 times. This is because *ScanMapRange* calls the $C()$ function for too many times on 3-dimensional data while *RoughMapRange* does not call the $C()$ function at all. *GetNextZ* does not call the $C()$ function explicitly, but it requires some computation that is similar to the $C()$ function.

Fig. 16 shows the page access performance. *RoughMapRange* has slightly larger numbers of page accesses because it relaxes the process of generating the key ranges. However, we have seen that *RoughMapRange* has a much less overall query response time, this is because its low computation cost outweighs the overhead incurred by a smaller number of extra page accesses. *GetNextZ* generates the same index key ranges as *ScanMapRange* does and hence the numbers of page accesses for these two algorithms are the same.

2) Next we compare *EdgeMapRange* with *ScanMapRange* and *GetNextH* on the Hilbert curve. In this set of experiments we only record the response time since the three methods have the same numbers of page accesses because they differ in the process of generating the key ranges for the B^+-tree search but generate the same key ranges.

On 2-dimensional data the three *MapRange* algorithms show very similar performance; when the query selectivity increases, *EdgeMapRange* gradually shows its advantage. On 3-dimensional data, *EdgeMapRange* outperforms *ScanMapRange*



(a) uniform 2D data set (b) uniform 2D data set (c) uniform 2D data set

(d) uniform 2D data set (e) uniform 2D data set (f) uniform 2D data set

Fig. 17. Window query performance of the standalone index implementations, uniform 2D data

by 2 times on average. The advantage of *EdgeMapRange* is due to the fact that it computes fewer curve values. *GetNextH* also computes fewer curve values, but it takes more time for each computation. We omit the detail figures due to the journal page limit.

7.2 Performance of SSI as a Standalone Implementation

In this subsection we evaluate the performance of various indexing methods as standalone implementations. The experiments consider 10 methods: SSI with Z-curve (denoted by "SSI-Z"), SSI with Hilbert-curve (denoted by "SSI-H"), B^{dual}-tree (denoted by "B-dual"), Sequential Scan (denoted by "Scan"), the R*-tree (denoted by "R*-tree"), Dual-transform with space-filling curves of orders from 2 to 5, (denoted by "DT-2" to "DT-5", respectively) and Size Separation Spatial Join (denoted by "SSSJ").

7.2.1 *Experiments on 2-dimensional Data Sets.* Fig. 17 shows the query performance on 2-dimensional uniform data where we vary the data set cardinality. The response time of SSSJ, B-dual and other methods are presented in three figures, Fig. 17a, Fig. 17b and Fig. 17c, respectively, rather than in the same figure because they are in very different ranges. From these figures we can see that SSSJ performs the worst. Its response time is in tens of milliseconds for a data set with only 25,000 objects. Considering that in real systems there are commonly tens of thousands of users performing queries at the same time, this response time is too slow to provide satisfactory user experience. B-dual, Scan, DT-3 and DT-5 are slightly faster (response time of DT-2 and DT-4 are between DT-3 and DT-5 and hence are omitted). SSI-Z and SSI-H outperform all the above methods significantly. Their response time are around 1 millisecond for the data set of 1 million objects. R*-tree is the closest to the SSI methods. Its response time is around 10 millisecond for

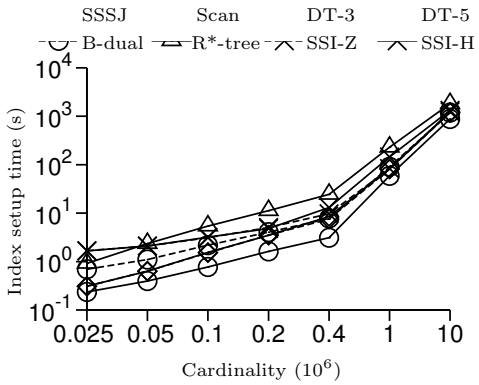


Fig. 18. Setup time of the standalone index implementations, uniform 2D data

the data set of 1 million objects.

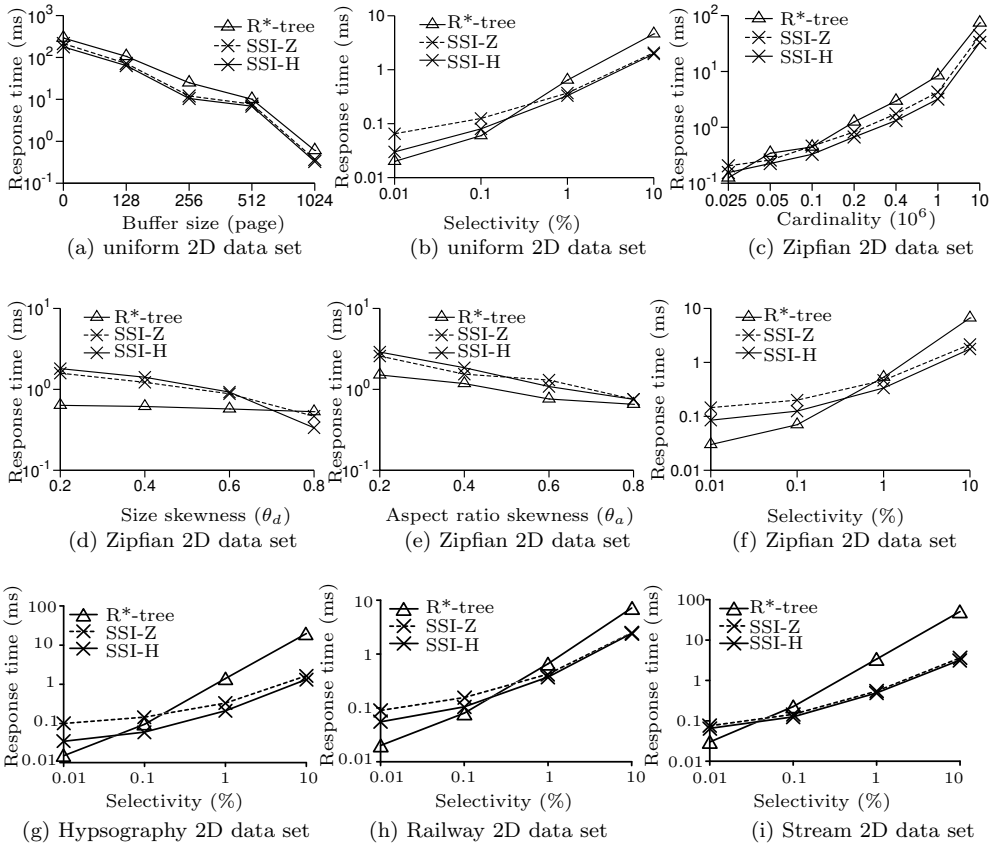
We show the corresponding numbers of page accesses of these methods in Fig. 17d, Fig. 17e, and Fig. 17f, respectively. The SSI methods also show good performance in page access, although the scale of performance difference in page access is different from that in response time. The R*-tree shows the smallest number of page accesses, while the SSI methods' page access numbers are very close to that of the R*-tree.

We show the index setup time in Fig. 18, where no bulk loading is used for any index. As expected the methods with high query processing time have relatively low index setup time. Specifically, Scan does not require any setup time at all; SSSJ, B-dual and the DT methods can all index 1 million objects in around 80 seconds. SSI-Z and SSI-H are slightly slower, with 90 and 144 seconds to build an index on 1 million objects; R*-tree is the slowest, which requires 229 seconds to do the same. Considering the high performance gain in the query processing time, and that the index setup is only performed once for a long time, the slightly higher setup cost of the SSI methods is worth while.

Since we focus on getting low query response time, in the following experiments, we omit presenting the page access and index setup time results to keep the paper concise, as they have similar behavior to those of the above experiments. Also, since the other methods are highly uncompetitive, we will only show the results of R*-tree, SSI-H and SSI-Z for the following set of experiments.

Fig. 19 shows the query processing time on other 2-dimensional data sets and parameter settings. Specifically, in Fig. 19a we vary the number of buffer pages from 0 to 1024. In this set of experiments the number of index pages of R*-tree, SSI-Z and SSI-H are 682, 869 and 838, respectively, and the percentage of the index buffered ranges from 0% to 100% for each method. In Fig. 19b we vary the window query selectivity on the uniform data set. From Fig. 19c to Fig. 19f we vary the cardinality, the skewness of the object size distribution, the skewness of the object aspect ratio distribution and the window query selectivity on the Zipfian data sets. From Fig. 19g to Fig. 19i we vary the window query selectivity on real data sets.

We see from these figures that the SSI methods outperform the R*-tree in most cases. Only when the query selectivity is very small (i.e., less than 0.1%) or the object size is very skew that the R*-tree shows a smaller response time (cf. Fig. 19b, Fig. 19d and Fig. 19e). Meanwhile, the SSI methods show relatively steady perfor-



(a) uniform 2D data set (b) uniform 2D data set (c) Zipfian 2D data set

(d) Zipfian 2D data set (e) Zipfian 2D data set (f) Zipfian 2D data set

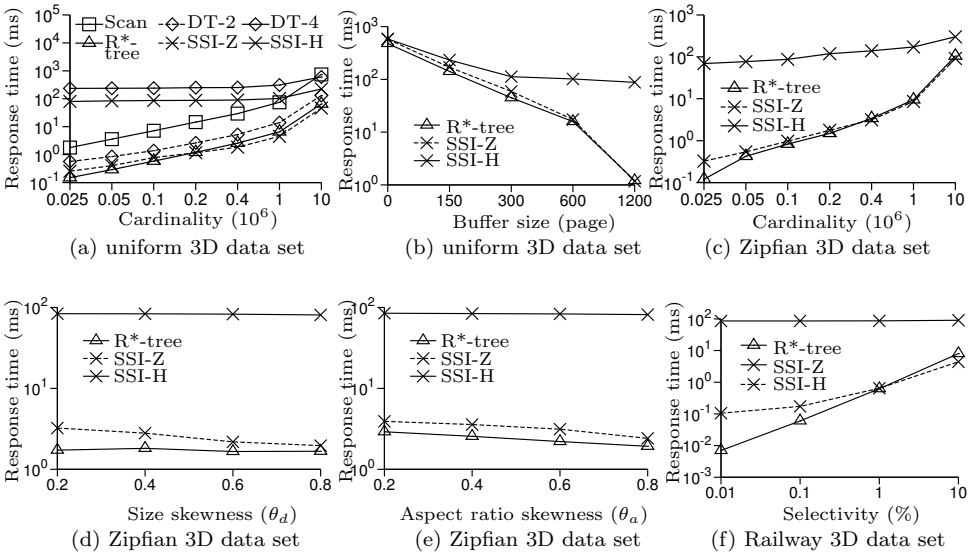
(g) Hypsography 2D data set (h) Railway 2D data set (i) Stream 2D data set

Fig. 19. Window query performance of the standalone index implementations, other 2D data

mance when the experimental parameter values are varied. This demonstrates the robustness of the SSI methods.

7.2.2 *Experiments on 3-dimensional Data Sets.* Fig. 20 shows the query processing time on 3-dimensional data sets. Among the various methods, SSSJ and B-dual perform as bad as they did in the experiments on the 2-dimensional data sets and hence are omitted. DT-5 cannot finish after hours. DT-5 is so slow because it involves curve value mapping of a space-filling curve of order 5 in a 6-dimensional data space. Therefore, it is not included in the results.

Fig. 20a shows the result where the data set cardinality is varied. We can see that, like the experiments above, SSI-Z and R*-tree have the smallest response time and they both significantly outperform other methods. DT-2 is close while DT-4 is the worst. The response time of DT-3 is between DT-2 and DT-4 and hence is omitted. We observe that SSI-H shows less competitive performance. This is because of the relatively high complexity of curve value mapping on the Hilbert-curve in 3-dimensional space, which dominates the query processing cost. We also see that the response time of SSI-H is quite stable when the data set cardinality increases. As a result, it becomes closer to the response time of the other methods



(a) uniform 3D data set (b) uniform 3D data set (c) Zipfian 3D data set

(d) Zipfian 3D data set (e) Zipfian 3D data set (f) Railway 3D data set

Fig. 20. Window query performance of the standalone index implementations, 3D data

as the data set cardinality increases.

In Fig. 20b we vary the number of buffer pages from 0 to 1200. Since in this figure the number of index pages of R*-tree, SSI-Z and SSI-H are 977, 1131 and 1128, respectively, the percentage of the index buffered ranges from 0% to 100% for each method. We observe that the response time decreases for all methods as more buffer pages are used. SSI-Z and R*-tree show similar performance constantly, while SSI-H also shows similar performance to R*-tree when fewer buffer pages are used, where the I/O time becomes dominating in the query processing cost.

From Fig. 20c to Fig. 20e we vary the cardinality, the skewness of the object size distribution and the skewness of the object aspect ratio distribution on the Zipfian data sets. The comparative performance of SSI-Z, SSI-H and R*-tree is similar to that on uniform data sets.

In Fig. 20f we vary the query selectivity on real data. Again, R*-tree and SSI-Z show close performance and SSI-Z performs better when the query selectivity gets large (i.e., larger than 1%). This is because when the query selectivity increases, the tree-based pruning of R*-tree degrades quickly.

7.2.3 *Summary.* We summarize the above experimental results as follows.

—The SSI methods outperform other mapping based indexing methods by orders of magnitude in various data sets and experimental settings in terms of both response time and page access.

—The SSI methods show competitive performance when compared with R*-tree. They outperform R*-tree for most cases (i.e., when the data set cardinality is larger than 50,000 or the query selectivity is larger than 0.1%) on 2-dimensional data. For 3-dimensional data, SSI-Z has very similar performance to R*-tree for most cases and outperforms R*-tree when the data set cardinality is large; SSI-H

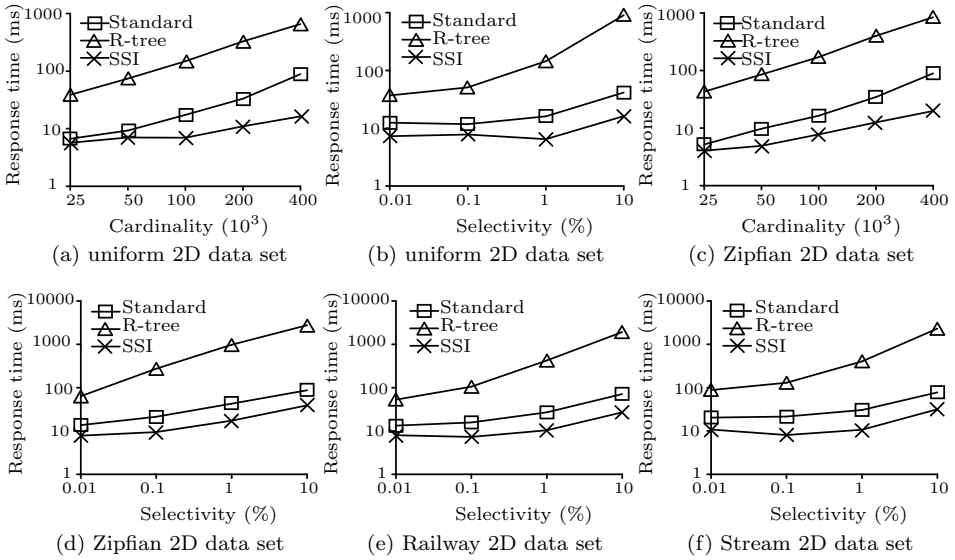


Fig. 21. Window query performance of indexes in the PostgreSQL implementation

shows less competitive performance but is still close to SSI-Z where the data set cardinality or window query selectivity is large.

7.3 Performance of SSI as Integrated in a DBMS

To evaluate the performance of SSI on full-fledged DBMSs, we implement SSI on two DBMSs. One is from the research community, PostgreSQL, and the other is a commercial DBMS.

On each DBMS we evaluate the performance of three indexing methods:

—*SSI.* DBMS implementation of SSI as explained in Section 6.

—*Standard.* The DBMS' standard data indexing component, where the objects are stored in a database table as described in Section 6. But instead of creating a B^+-tree index on the SSI keys, this method creates one B^+-tree index on each bound of the object MBRs (i.e., one B^+-tree for each MBR bound column of the database table). When a window query arrives (in the form of a SQL query like the one shown in Section 6, but without the predicates on the SSI keys), the DMBS automatically chooses the proper B^+-tree to evaluate the query.

—*R-tree.* The DBMS' built-in spatial component (specifically, the R-tree).

7.3.1 *PostgreSQL.* As shown in Fig. 21, when implemented on PostgreSQL, SSI outperforms the R-tree and PostgreSQL's standard data indexing component constantly. Compared with PostgreSQL's standard data indexing component, SSI is more than twice as fast on average (please note the logarithmic scale). The response time of SSI grows much slower as the increase of data set cardinality. We observe that the R-tree in PostgreSQL shows even worse performance than the standard data indexing component does. We look into the query processing procedure of PostgreSQL when the R-tree is used for indexing and find that the

query optimizer is more likely to use a sequential scan to process window queries rather than using the R-tree index. A possible reason is that PostgreSQL's query optimizer has not got accurate estimation on the cost of using the R-tree index and thus, it is being conservative in most cases and uses sequential scans instead. We find that this is actually a common phenomenon observed by many users in the PostgreSQL community.

7.3.2 *A commercial DBMS.* We create an R-tree in the commercial DBMS to index the spatial objects following the official documentation. SSI again outperforms the R-tree index as well as the standard index used in the commercial DBMS when the data set cardinality and the query selectivity are varied. We observe that when the data are more skewed, the advantage of SSI is more significant (by an order of magnitude; we omit the figures due to the journal page limit). This is because of the cumulative mapping used to generate an approximately uniform distribution to reduce the false positives in the filtering stage of query processing.

8. CONCLUSION

In this paper we presented SSI, a mapping based index scheme for window queries on spatial objects with non-zero extents. It uses size separation, cumulative mapping, window query expansion and novel *MapRange* algorithms to overcome the drawbacks of existing mapping based indexes with extensive experiments using both real and synthetic data sets. We show that:

—SSI outperforms other mapping based indexing methods by orders of magnitude. As a standalone implementation, SSI is competitive with the R*-tree.

—SSI can be easily implemented on a DBMS and it outperforms the DBMS integrated R-tree implementation from two off-the-shelf DBMSs by up to three orders of magnitude in processing window queries.

—Our cost model to optimize the performance of SSI is highly accurate under various settings.

Acknowledgements

This work is supported by the Australian Research Council's Discovery funding scheme (project number DP130104587). Rui Zhang is supported by the Australian Research Council's Future Fellow funding scheme (project number FT120100832).

REFERENCES

Aref, W. G. and Ilyas, I. F. 2001. Sp-gist: An extensible database index for supporting space partitioning trees. *J. Intell. Inf. Syst. 17,* 2-3, 215–240.

Arya, M., Cody, W. F., Faloutsos, C., Richardson, J., and Toya, A. 1994. Qbism: Extending a dbms to support 3d medical images. In *ICDE*. 314–325.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*. 322–331.

Bentley, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18,* 9, 509–517.

Berchtold, S., Böhm, C., and Kriegel, H.-P. 1998. The Pyramid-technique: Towards breaking the curse of dimensionality. In *SIGMOD*. 142–153.

Böhm, C., Klump, G., and Kriegel, H.-P. 1999. Xz-ordering: A space-filling curve for objects with spatial extension. In *SSD*. 75–90.

Butz, A. 1971. Alternative Algorithm for Hilbert's Space-Filling Curve. *IEEE Transactions on Computers 100,* 20, 424–426.

Comer, D. 1979. The ubiquitous b-tree. *ACM CS 11,* 2, 121–137.

Faloutsos, C. and Rong, Y. 1991. Dot: A spatial access method using fractals. In *ICDE*. 152–159.

Faloutsos, C. and Roseman, S. 1989. Fractals for secondary key retrieval. In *PODS*. 247–252.

Finkel, R. A. and Bentley, J. L. 1974. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica 4,* 1, 1–9.

Gaede, V. and Günther, O. 1998. Multidimensional access methods. *ACM CS 30,* 2, 170–231.

Google. 2012. Google search. http://en.wikipedia.org/wiki/Google_Search.

Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*. 47–57.

Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. 1995. Generalized search trees for database systems. In *VLDB*. 562–573.

Jagadish, H., Ooi, B. C., Tan, K.-L., Yu, C., and Zhang, R. 2005. iDistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *TODS 30,* 2, 364–397.

Jensen, C. S., Lin, D., Ooi, B. C., and Zhang, R. 2006. Effective density queries on continuously moving objects. In *ICDE*. 71.

Koudas, N., Ooi, B. C., Tan, K.-L., and Zhang, R. 2004. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB*. 804–815.

Koudas, N. and Sevcik, K. C. 1997. Size separation spatial join. In *SIGMOD*. 324–335.

Lawder, J. K. and King, P. J. H. 2001. Querying multi-dimensional data index using the hilbert space-filling curve. *SIGMOD Rec. 30,* 1, 19–24.

Microsoft. 2008. Microsoft sql server 2008. http://www.microsoft.com/sqlserver/2008/en/us/spatial-data.aspx.

Moon, B., Jagadish, H. V., Faloutsos, C., and Saltz, J. H. 2001. Analysis of the clustering properties of the hilbert space-filling curve. *TKDE 13,* 1, 124–141.

Nutanong, S., Zhang, R., Tanin, E., and Kulik, L. 2008. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB 1,* 1, 1095–1106.

Oracle. 2007. *Oracle Spatial Technologies*. http://download.oracle.com/otndocs/products/spatial/pdf/spatial_features_jsirev.pdf.

Orenstein, J. A. and Merrett, T. H. 1984. A class of data structures for associative searching. In *PODS*. 181–190.

Ramsak, F., Markl, V., Fenk, R., Zirkel, M., Elhardt, K., and Bayer, R. 2000. Integrating the ub-tree into a database system kernel. In *VLDB*. 263–272.

Stefanakis, E., Theodoridis, T., Sellis, T. K., and cheung Lee, Y. 1997. Point representation of spatial objects and query window extension: a new technique for spatial access methods. *International Journal of Geographical Information Science 11,* 6, 529–554.

Tao, Y., Papadias, D., and Sun, J. 2003. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*. 790–801.

Weber, R., Hans-J, S., and Blott, S. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*. 194–205.

Xu, P. and Tirthapura, S. 2012. On the optimality of clustering properties of space filling curves. In *PODS*. 215–224.

Yiu, M. L., Tao, Y., and Mamoulis, N. 2008. The b^{dual}-tree: indexing moving objects by space filling curves in the dual space. *VLDBJ 17,* 3, 379–400.

Zauhar, R. J., Moyna, G., Tian, L., Li, Z., and Welsh, W. J. 2003. Shape signatures: A new approach to computer-aided ligand and receptor-based drug design. *Journal of Medical Chemistry 46,* 26, 5674–5690.

Zhang, R., Kalnis, P., Ooi, B. C., and Tan, K.-L. 2005. Generalized multidimensional data mapping and query processing. *TODS 30,* 3, 661–697.

Zhang, R., Ooi, B. C., and Tan, K.-L. 2004. Making the pyramid technique robust to query types and workloads. In *ICDE*. 313–324.