Zoned-partitioning of tree-like access methods

Seon Ho Kim\textsuperscript{a,*}, Byunggu Yu\textsuperscript{b}, Jae-young Chang\textsuperscript{c}

\textsuperscript{a}Computer Science Department, University of Denver, 2360 S. Gaylord Street, Denver, CO 80208, USA
\textsuperscript{b}Computer Science and Information Technology, University of the District of Columbia, Washington, DC 20008, USA
\textsuperscript{c}Computer Engineering Department, Hansung University, Seoul, South Korea

Received 16 May 2007; received in revised form 6 November 2007; accepted 7 November 2007
Recommended by F. Carino Jr.

Abstract

The performance of access methods and the underlying disk system is a significant factor in determining the performance of database applications, especially with large sets of data. While modern hard disks are manufactured with multiple physical zones, where seek times and data transfer rates vary significantly across the zones, there has been little consideration of this important disk characteristic in designing access methods (indexing schemes). Instead, conventional access methods have been developed based on a traditional disk model that comes with many simplifying assumptions such as an average seek time and a single data transfer rate. The paper proposes novel partitioning techniques that can be applied to any tree-like access methods, both dynamic and static, fully utilizing zoning characteristics of hard disks. The index pages are allocated to disk zones in such a way that more frequently accessed index pages are stored in a faster disk zone. On top of the zoned data placement, a localized query processing technique is proposed to significantly improve the query performance by reducing page retrieval times from the hard disk.

\textcopyright{} 2008 Elsevier B.V. All rights reserved.

Keywords: Partitioning; Disk zoning; Access methods; Tree-like index structures; Query processing; Database management

1. Introduction

In most secondary storage access methods, the index structure is a hierarchy of index pages that form a balanced tree with a single root. For example, in a spatial access method, the \((d \geq 1)\)-dimensional data space can be recursively divided into \textit{hyper-rectangles} at every level of the structure. The rectangle of the \textit{root page} encloses the entire data set. When an index search is requested, it incurs a sequence of page accesses. Thus, the performance of any index searches depends on the number of page accesses. In practice, the performance of an access method is measured with its underlying disk system, so that its disk I/O time per page access must be considered as well as the number of page accesses.

Disk I/O time depends on the physical characteristics of underlying disk subsystem. Modern hard disks are manufactured with \textit{zoned recording}, which groups adjacent disk cylinders into zones \([1,2]\). Tracks are longer towards the outer portions of a disk platter as compared to the inner portions.
Hence, more data can be recorded in the outer tracks when the maximum linear density, i.e., bits per inch, is applied to all tracks. The results are multiple physical zones in a disk, where seek times and data transfer rates vary significantly across the zones. However, there is a marked lack of investigation on how to optimize access methods given a zoned disk model. Instead, conventional access methods have been developed based on a traditional disk model (single zone disk model) that comes with many simplifying assumptions such as an average seek time and a single data transfer rate.

When index pages are intensively accessed, e.g., index search, the effective disk I/O time also depends on the locations of pages within the disk. It is well known that accessing pages in a small portion of a disk (i.e., localized access or partitioning) is much faster than accessing widely scattered pages.

Thus, the main proposition of this paper is that the performance of any tree-like access method can be improved when each index search can be localized as follows: (1) the index structure of the access method is split into disjoint parts, each of which is stored in a specific disk zone in its entirety, (2) at every point in time, all running threads are accessing only one zone and a thread accesses another zone only after it completes all required accesses to the current zone. That is, each search is a sequence of zone-bursts, where a zone-burst is a sequence of all necessary page accesses to a certain zone. The performance enhancement comes from the fact that accessing two pages located within the same physical zone is faster than accessing two pages in different zones, due to minimized seek times. We call it “zoned-partitioning” to place an index page into a specific disk zone to enhance the performance of index search.

Zoned-partitioning can significantly enhance the performance of the access method by harnessing and combining the following properties: the access frequencies of index pages and the variable data transfer rates of the underlying multi-zone disk. Considering a tree-type index structure and random queries, one can think of approximating the access frequency of each page by means of the level of the page in the index structure. For example, the root page of an index structure has the highest access frequency among the pages constituting the index structure, since every search must access the root. In this sense, the leaf level pages are associated with the lowest access frequency. Then, the proposed zoned-partitioning technique can assign frequently accessed pages to a faster zone for a faster access. The level of a page in the index structure determines the zone in which the page is stored. This is a simple and straightforward approach that can be applied to any tree-like index structures and is the basis of the idea of pinning a few top levels of an active index structure in the database buffer space [3]. However, this approach ignores the underlying data distribution, and all queries that produce a non-empty result must access all the zones. In fact, this approach is based on the assumption that all pages at the same level are equally important and have the same likelihood of being accessed by a random query. In practice, this is not the case.

For each unknown random range query, a page that has a larger range in the data space is more likely accessed. Considering non-uniformly distributed data objects, it is often the case with any index structure that a page has a larger range in the data space than an upper level page that is not a direct ancestor. In addition, when the query distribution follows the underlying data distribution, a page that encloses more data objects in the data space has a higher access probability. Moreover, in many applications, the queries have no “affinity” for certain dimensions or the query patterns often change over time. In this case, all dimensions are equally important, and a page that has a smaller perimeter in the data space tends to have a lower average access frequency. Therefore, the access frequency of each page should be determined by the actual region of the page in the data space and, optionally, the number of data objects that the page encloses. Consequently, the placement of a page on to a specific disk zone can be determined by its access frequency.

This paper proposes a novel optimization technique that can be applied to a variety of one- to high-dimensional access methods, both static and dynamic, and that can fully exploit the zoning characteristics of modern hard disks. In the proposed approaches, index structures are zoned in such a way that more frequently accessed index pages are stored in a faster disk zone. Working with the zoned-partitioning, our localized query processing technique which implements the zone-burst significantly improves the query performance by minimizing page retrieval times from the hard disk. The idea of zoned-partitioning was introduced in our previous work as a partial solution for special environments [4]. This study has augmented and integrated our
previous work as a generalized framework, including far more comprehensive experiments with large data sets, both synthetic and real, under various environments. For a focused discussion, this paper considers only a single disk case. However, the techniques proposed in this paper can be generalized to multi-disk (e.g., RAID) systems, especially in the presence of underlying intelligent disk array systems that can logically construct virtual storage volumes (such as 3PAR Inserv storage system [5]).

The rest of this paper is organized as follows. Section 2 presents some backgrounds and defines the problem. Section 3 defines our novel zoned-partitioning techniques for tree-like access methods. Then the localized query processing algorithm is described in Section 4. In Section 5, we discuss our extensive experiments based on both synthetic data and real data. Conclusions and future research directions follow in Section 6.

2. Preliminaries

2.1. Multi-zone disks

One of the most important physical characteristics of a modern magnetic disk drive is its zones: a disk consists of several zones, each providing a different storage capacity and transfer rate. Zoned recording is an approach utilized by disk manufacturers to increase the storage capacity of various types of disks [1]. This technique groups adjacent disk cylinders into zones [1,2]. Tracks are longer towards the outer portions of a disk platter as compared to the inner portions; hence, more data may be recorded in the outer tracks when the same maximum linear density (i.e., bits per inch) is applied to all tracks. A zone is a contiguous collection of disk cylinders whose tracks have the same storage capacity, i.e., the number of sectors per track is constant in the same zone. Hence, outer zones have more sectors per track than inner zones.

Different disk models have different numbers of zones (e.g., Seagate Cheetah X15 has nine zones; Seagate Barracuda 7200.7 has 15 zones). Different zones provide different transfer rates because: (1) the storage capacity of the tracks for each zone is different, and (2) the disk platters rotate at a fixed number of revolutions per second. Each disk zone consists of one or more disk cylinders and all cylinders in the same zone have the same data transfer rate (the same maximum data transfer rate).

The effective data transfer rate when a disk is accessed in practice is different from the maximum rate due to involved overhead. Thus, the actual access time of an index page consists of a seek time, a rotational latency, and a page transfer time. A rotational latency is determined by the disk revolution time and a seek time is a function of the distance between two locations on a disk, for example, \( x \) cylinders apart. Table 1 shows an example of a multi-zone disk.

<table>
<thead>
<tr>
<th>Zone no.</th>
<th>Size (GB)</th>
<th>Max rate (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>57.5</td>
</tr>
<tr>
<td>1</td>
<td>3.5</td>
<td>55.4</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
<td>54.7</td>
</tr>
<tr>
<td>3</td>
<td>4.0</td>
<td>52.7</td>
</tr>
<tr>
<td>4</td>
<td>3.0</td>
<td>50.6</td>
</tr>
<tr>
<td>5</td>
<td>2.5</td>
<td>48.1</td>
</tr>
<tr>
<td>6</td>
<td>3.0</td>
<td>45.6</td>
</tr>
<tr>
<td>7</td>
<td>2.5</td>
<td>43.6</td>
</tr>
<tr>
<td>8</td>
<td>2.5</td>
<td>41.9</td>
</tr>
</tbody>
</table>

The effective data transfer rate when a disk is accessed in practice is different from the maximum rate due to involved overhead. Thus, the actual access time of an index page consists of a seek time, a rotational latency, and a page transfer time. A rotational latency is determined by the disk revolution time and a seek time is a function of the distance between two locations on a disk, for example, \( x \) cylinders apart. Table 1 shows an example of a multi-zone disk.
index point data. Fig. 1 shows an example of a real R*-tree index structure for point data.

An index structure is built and updated in two ways: (1) static-updating (also called as bulk-updating) and (2) dynamic updating. In the latter case, each data object is inserted or deleted at runtime. An efficient dynamic update algorithm for an index structure is required when the index structure is used in an operational database in a transactional environment.

In recent years, efficient bulk-updating algorithms are required in many database applications where static index structures are built on a populated data set that is seldom or periodically updated. For example, in many databases for analytical tasks (e.g., data warehouses and data marts), index structures are built on an already populated data set. Each update (if there is) typically inserts or deletes a relatively large number of data objects. Bulk-updating is based on a bulk-loading or packing algorithm that can build or reorganize an index structure on an existing data set. A bulk-loading algorithm is typically designed for a specific index structure. For example, a bulk-loading algorithm for B-trees (one-dimensional tree-like access methods) and a bulk-loading algorithm for R-trees can be found in Refs. [6] and [7–9], respectively. In general, bulk-loading algorithms produce a rather optimized index structure due to the fact that bulk-loading algorithms start with a whole data set. However, before our research that this paper represents, no other investigation has been done on how to optimize access methods given a multi-zone disk model.

2.3. Problem description

For partitioning an index structure, a specific set of the disk zones are first selected. These selected zones can represent faster disk zones for frequently used index structures, or underutilized disk zones, or both, depending on effective system administration policies. Note that selecting all the zones for every index structure or any specific index structure is certainly not prohibited. The rest of this section describes the zoned-partitioning problem and relevant issues.

An optimal zoned-partitioning algorithm should satisfy the following goals given a set of selected disk zones and an index structure:

- G1: For any index page \( p \) stored in zone \( zid \), there is no index page \( p' \) in zone \( zid' \) such that \( zid' \) is slower than \( zid \) and the access probability of \( p' \) is greater than that of \( p \).

- G2: All selected zones for the index structure is evenly utilized.

To achieve G1, it is required to accurately compute the access frequency of a page when it is inserted to the index. A list of placement of existing pages with their access frequencies should also be maintained. The purpose of G2 is to avoid the situation that a zone becomes full while others have plenty of space because it will be difficult to achieve G1 if a page is assigned to a zone that is already full. Thus, pages should be evenly distributed.
across participating zones. Note that it may be unavoidable to successively reallocate multiple pages to enforce an even utilization of zones.

First, we need to define the access frequency (or probability) of a page in a given index structure. Let us start with two functions: Measure\(_p\) is a function that returns either area or margin of a page \(p\); Pop\(_p\) is a function that returns the number of objects contained by the leaf level of the sub-tree rooted at \(p\) (i.e., the number of the data objects within the region of \(p\)). As discussed in Section 1, the zone where a page \(p\) to be stored is determined by Measure\(_p\) and Pop\(_p\). To avoid any possible confusions and misunderstandings, we first define area and margin as given in Definitions 1 and 2.

**Definition 1.** The area of an index page that is associated with a \((d\geq 1)\)-dimensional hyper-rectangle \(R\) in the data space is defined as \(\prod_{i=1}^{d}(h_{i} - l_{i})\) where, for all \(i = 1, ..., d, h_{i}\) and \(l_{i}\) are the high- and low-endpoints of \(R\) along dimension \(i\).

**Definition 2.** The margin of an index page that is associated with a \((d\geq 1)\)-dimensional hyper-rectangle \(R\) in the data space is defined as \(2^{d-1}\sum_{i=1}^{d}(h_{i} - l_{i})\) where, for all \(i = 1, ..., d, h_{i}\) and \(l_{i}\) are the high- and low-endpoints of \(R\) along dimension \(i\).

Now, let Measure\(_U\) and Pop\(_U\) be the area (or margin) of the root page and the total number of the data objects, respectively. Then one can consider Eq. (1) to be a weighted sum estimating the access probability of a page \(p\) for an unknown query. We call this the Relative Importance (RI) of \(p\):

\[
RI(p) = W1 \times \frac{\text{Measure}(p)}{\text{Measure}\_U} + W2 \times \frac{\text{Pop}(p)}{\text{Pop}\_U},
\]

where \(0 \leq W1 \leq 1\), \(0 \leq W2 \leq 1\), and \(W1 + W2 = 1\).

Considering a large database, the estimated access probability of a leaf page can be very small (i.e., close to 0) while the root page is always accessed (i.e., the access probability is 1). If the distribution of the queries does not follow the distribution of the data, \(W1\) becomes 1 and \(W2\) becomes 0. We assume this through the rest of this paper. Then, RI\(_p\) in Eq. (1) can be simplified as follows:

\[
RI(p) = \begin{cases} 
\frac{\text{Measure}(p) - \text{MinM}}{\text{MaxM} - \text{MinM}} & \text{if MaxM} \neq \text{MinM}, \\
1 & \text{otherwise}.
\end{cases}
\]

where Measure\(_p\), Min\(_M\), and Max\(_M\) is the area (or margin) of an index page \(p\), the minimum recorded value of Measure, and the maximum recorded value of Measure (note that Max\(_M\) = Measure\(_\text{root}\)) in the given index tree, respectively.

Now, for further discussion, we introduce the following terms that frequently used in the paper. Let \(NZ\) and Cap be the total number of the selected zones to store a given index and the total storage capacity of the selected zones, respectively. A unique zone identification number is assigned to each zone in such a way that a smaller logical zone number is always associated with a faster zone (in terms of page read/write time using effective transfer rate in the zone). Thus, zone 0 represents the fastest zone among the given \(NZ\) zones. \(Z\text{Cap}[zid]\) represents the storage capacity of zone \(zid\). Then, the relative portion of \(Z\text{Cap}[zid]\) to the total storage capacity Cap can be defined as \(ZR[zid] = Z\text{Cap}[zid]/\text{Cap}\), for all \(zid = 0, ..., NZ-1\). In the case of ideal even distribution of TS pages across \(NZ\) zones, \(ZR[zid] \times TS\) pages are assigned to each zone \(zid\), where \(TS\) is the total number of pages constituting the given index structure. However, in practice, the number of pages assigned to a zone can be different from the ideal case.

Assuming a dynamic update environment, a zoning approach (say \(\text{Naive}\)) should be able to assign pages to appropriate zones while satisfying both goals. First, all pages are sorted in a descending order with RI value. Then, a function rank\(_p\) returns the rank of a page \(p\) (rank\(_p\) = 1, ..., TS). Using rank\(_p\), one can define a function to represent the percentile rank of \(p\) among \(TS\) pages: pctrank\(_RI\)(\(p\)) = \((TS - \text{rank}(p) + 1)/TS\). Then, pctrank\(_RI\)(\(p\)) represents the relative size of RI\(_p\) using a value between 0 and 1. It has the same meaning of the function provided in Excel and SQL server. For example, pctrank\(_RI\) (root) has the value of 1 and leaf pages have near zero values. Using pctrank\(_RI\)(\(p\)), the ideal location of a page \(p\) in a disk can be zone \(k\) that satisfies the following: if pctrank\(_RI\)(\(p\))
\[ ZR[NZ-1] \text{ then } k = NZ-1. \text{ Otherwise, } k \text{ is determined using Eq. (3):} \]
\[ \sum_{i=0}^{NZ-k-2} Z[R[NZ-i-1] < pctrank RI(p) \]
\[ \leq \sum_{i=0}^{NZ-k-1} Z[R[NZ-i-1]. \quad (3) \]

Then, a searching technique to find the ideal location of page \( p \) can be implemented in a straightforward way using Eq. (3). However, in reality, the *Naive* approach may face the following two critical problems:

1. The value of \( pctrank RI(p) \) for all \( p \) in an index should be known for the assignment of pages across zones. To do so, one may need to maintain a separate data structure to store the sorted list of \( RI \) values of all stored pages and need to recalculate \( pctrank RI(p) \). The maintenance of such data structure can create a significant overhead.
2. Even though the above overhead can be tolerated, there could be a more serious problem in a dynamic environment where insertion and deletion of pages frequently happen. Insertions and deletions can dynamically change the utilization of individual zone. For a perfectly even distribution, lots of pages might be moved to new zones in the fashion of chain reaction. This may create intolerable overhead in some environments such as online transaction processing.

These overhead problems make the *Naive* approach inefficient for many practical applications. Thus, this paper focuses on the development of sophisticated zoned-partitioning algorithms that (1) work with a less overhead (with minimum amount of extra information), (2) target a close approximation of even distribution for a faster performance (rather than a perfectly even distribution) while trying to achieve \( G1 \) and \( G2 \) as much as possible, and (3) make the zone-burst operation possible.

3. Zoned-partitioning algorithm

3.1. The algorithm

First, we need to determine zone \( k \) to assign a given page \( p \). It can be obtained using Eq. (3), which requires an accurate \( pctrank RI(p) \) value. However, as explained above, it requires an extra overhead. An alternative way of calculating \( pctrank RI(p) \) is to use an estimation, i.e., one can introduce a simplifying assumption that \( pctrank RI(p) \) and \( RI(p) \) have a linear relation. In reality, \( pctrank RI(p) \) and \( RI(p) \) are very different. For example, most pages in an index are leaf pages at the bottom so most \( RI \) values are close to zero. The distribution of actual \( RI \) values is concentrated near zero. This implies that \( pctrank RI(p) \) are greater than \( RI(p) \) in most pages. Thus, if one simply replaces \( pctrank RI(p) \) with \( RI(p) \), most pages will be assigned to slow zones due to near-zero \( RI(p) \) values. To resolve the problem and evenly utilize zones, the page assignment should consider the zone utilization indicator \( ZUI \). When a zone is over-utilized, there should be a way to allocate pages in other zones.

General zoned-partitioning works as follows. Given a page \( p \), \( RI(p) \) is calculated using Eq. (2). After substituting \( pctrank RI(p) \) with \( RI(p) \) in Eq. (3), the appropriate zone \( k \) that satisfies Eq. (3) is determined. If zone \( k \) is not over-utilized, \( p \) is assigned to the zone. Otherwise, the algorithm successively searches for the next faster zone than \( k \) to find a place for \( p \). This can continue until the algorithm reaches the fastest zone. This guarantees that a page is assigned to a zone appropriate for its \( RI \) value or a faster zone. At the same time, the utilization of zones can be relatively close to even. Thus, it supports the goals of zoned-partitioning described in Section 2.3.

Another important point that the algorithm should have is that it must make the zone-burst possible. For the guaranteed zone-burst, the following condition should be satisfied: when \( zone(i) \) is the zone where page \( i \) is stored, \( zone(i) \) should be less than or equal to \( zone(j) \) if page \( i \) is an ancestor of page \( j \). To satisfy the above condition, a page \( p \) should be assigned to a faster zone than \( p \)'s children. Considering all the above discussion, our zoned-partitioning algorithm, *Index_Zoning*, is defined in Fig. 2.

The algorithm consists of three parts. For a given index page \( p \), Part I calculates the relative importance, \( RI(p) \). Part II searches for the fastest zone \( (\text{maxchildzone}) \) among zones where page \( p \)'s children reside. Part III decides the most appropriate zone for page \( p \) using \( RI \), \( ZUI \), and \( \text{maxchildzone} \).

3.2. Discussions on the algorithm *Index_Zoning*

3.2.1. Successive reallocation of pages to guarantee zone-burst

In fact, *Index_Zoning* cannot complete the goals of this study because it lacks one of the most
important approaches, a guaranteed zone-burst. The algorithm only considers page \( p \)'s children, but not \( p \)'s ancestors. Thus, \( p \) might be assigned to a zone faster than the zones where \( p \)'s parents are stored. For a guaranteed zone-burst, the algorithm should be applied not only for the page \( p \) but also for the \( p \)'s ancestors toward the root of the index. There is no need to consider every ancestor of page \( p \). When a parent of page \( p \) can be found in the same or faster zone than the zone where page \( p \) can be stored, no more reallocation is necessary. The following algorithm can implement the successive reallocation problem using \textit{Index\_Zoning}.

The algorithm utilizes \textit{Index\_Zoning} to determine the zone number for the input page \( p \) and reallocates \( p \)'s parents to faster zones if they are stored in a slower zone than the \( p \)'s. This successive reallocation of parent pages may look like a significant overhead. However, it might not be a problem in reality because any tree-like index works in the same way. In a tree-like index, page insertion and modification happen at the bottom level and the change can be propagated towards the root. The successive update of any change is already an essential part of the index. Thus, by integrating the proposed algorithm in Fig. 3 into the insertion/modification algorithms of any given index, we can dynamically reallocate the index pages with minimal overhead.

3.2.2. Failure in searching an appropriate zone

There is one more consideration for \textit{Index\_Zoning} to be used in practice. As shown in Fig. 2, it returns in two different ways. It normally returns a zone number where the page \( p \) can be stored. However, it can return -1 when the search for an appropriate zone fails. A search may fail when even zone 0 cannot satisfy the last if-statement in Fig. 2. It might be possible not to be able to satisfy the condition, \( ZUI [0] \leq 1 \). That is, even zone 0 can be over-utilized. For example, there can be many pages with \( RI \) values close to 1, so all of them can be

---

**Algorithm Index\_Zoning**

**Input**: page \( p \)

**Output**: assigned zone number (0…NZ-1)

**Begin**

/* Part I: Dynamically calculate RI */

\[ Measure = \text{a Volume or a Margin of } p \]

\[ \text{if } Measure < \text{MinMeasure} \text{ then} \]

\[ \text{MinMeasure} = Measure; \]

\[ \text{end if} \]

\[ \text{if } Measure > \text{MaxMeasure} \text{ then} \]

\[ \text{MaxMeasure} = Measure; \]

\[ \text{end if} \]

\[ RI = (Measure-\text{MinMeasure}) / (\text{MaxMeasure}-Measure); \]

/* Part II: Find the fastest zone among those in which children of \( p \) are stored */

\[ \text{maxchildzone} = \text{NZ-1}; \]

\[ \text{for each child page } cp \text{ of page } p \text{ do} \]

\[ \text{if zone}(cp) > \text{maxchildzone} \text{ then} \]

\[ \text{maxchildzone} = \text{zone}(cp); \]

\[ \text{end if} \]

/* Part III: Find the zone in which \( p \) will be stored */

\[ \text{for (C\_ZR = 0, } i = 0; \{ <\text{NZ}; i++ \} \text{ do} \]

\[ \text{ZUI} = ZC[\text{NZ-1}] / (TS*ZR[\text{NZ-1}]); \]

\[ C\_ZR = C\_ZR + ZR[\text{NZ-1}]; \]

\[ \text{/* RI is an approximation of } p\text{ctrankRI(}p\text{)} */ \]

\[ \text{if } RI \leq C\_ZR \text{ and } ZUI \leq 1 \text{ and } \text{maxchildzone} \geq \text{NZ-i-1} \text{ then} \]

\[ \text{return } \text{NZ-i-1}; \]

\[ \text{end if} \]

/* Part III: Find the zone in which \( p \) will be stored */

\[ \text{return -1; /* fail */} \]

End

Fig. 2. General zoned-partitioning algorithm, \textit{Index\_Zoning}.
assigned to zone 0 resulting in an over-utilization of zone 0. Theoretically, it is possible even though it is unlikely in practice because the algorithm starts a search from slower zones due to the fact that \( pctrankRI(p) > RI(p) \) in most cases. It implies that it is unlikely that the fastest zone (zone 0) can be over-utilized earlier than other slower zones.

Nevertheless, one might think of two solutions when the problem happens. First, when it happens, one can immediately reorganize the entire pages for even utilization of zones that is one of the goals of this study. However, it will incur a huge overhead. An alternative way is to continue assigning pages to zone 0 even though \( ZUI[0] \) is greater than 1. Note that \( ZUI[0] = 1 \) does not mean the physical space of zone 0 is fully utilized. It only means that the assigned portion of zone 0 for zoned-partitioning has been saturated. The reorganization can be deferred until the physical capacity of zone 0 becomes full. In reality, this approach can be attractive because (1) most disks have a relatively large zone 0, (2) assigning more pages in zone 0 is not a bad idea considering zone 0 is the fastest zone in a disk.

### 3.3. Zoned-partitioning for static indexing environment

So far, we have assumed a general tree-like index structure where the data set is dynamically and frequently updated, i.e., pages are independently inserted or updated one by one. The algorithm has been discussed under the assumption.

Now, assume a static environment where an index structure is built or updated all together with a given complete set of data (for example, using a bulk-updating algorithm). In general, there can be two approaches in building a static index structure. One is to independently create each page and store in the disk one by one as described in Sections 3.1 and 3.2. The other is to create the entire index using a temporary space such as main memory and store the index in the disk all at once.

We can apply our zoned-partitioning algorithm for the static index structure. For the first approach, we can straightforwardly apply the \( \text{Index Zoning} \) because pages can be created and stored independently one by one. For the second approach, we can take advantage of the global information stored in main memory to more efficiently achieve the zoned-partitioning. For example, \( RI \) values of all pages are available in main memory. By sorting all pages based on their \( RI \) values, we can accurately calculate \( rank(p) \) and consequently the precise value of \( pctrankRI(p) \), rather than approximating the value from \( RI(p) \) as used in Section 3.1. Then the algorithm can ideally achieve the two goals of zoned-partitioning with minimum overhead. This is a special case of the general one when a large set of data is updated at once.

### 4. LQP: localized query processing

The proposed zoned-partitioning comes with the novel query processing technique called the \( \text{Localized Query Processing (LQP)} \). This section proposes this query processing technique.

A range query initiates a selection process that starts with the root page and propagates downward, traversing potentially multiple paths in the tree. At each interior page, the entries are tested to select the child pages that satisfy the query condition. In general, the query condition tests whether the region of the child page intersects with the \( \text{minimum bounding rectangle (MBR)} \) of the given query region. Whenever a leaf page is accessed, the procedure selects all resident objects or data entries that fall within the query region. The point query is a special case of range query, where the query range is infinitesimally small. SAMs provide several different types of range queries (topological queries) and the query processing procedure is more complicated. An in-depth coverage of this is found in Refs. [10,11].

An index search involves a sequence of page accesses. In order to take complete advantage of a zoned index structure in Section 3, each index search must be localized. That is, each search must be a sequence of zone-bursts, where a zone-burst is a

```
Algorithm Write_Index_Page
Input: page \( p \)
Output: None
Begin
    zone_number = Index_Zoning(p);
    write page \( p \) onto the zone zone_number;
    /* parent_of(p) is a function returning a parent page of \( p \) */
    while \((p \text{ is not a root}) \text{ and } \text{zone_number} < \text{zone(parent_of(p))}\) do
        \( p = \text{parent_of}(p) \);
        zone_number = Index_Zoning(p);
    end while
End
```

Fig. 3. Index page writing algorithm, \( \text{Write_Index_Page} \).
sequence of all necessary page accesses to a certain
zone. This is because of the fact that accessing two
pages located in two different zones is slower than
accessing two pages in the same zone due to an
increased seek time. Thus, the number of disk-head
moves across the disk zones must be minimized.
However, in a given tree, an index page
\( p \) can
possibly have a higher \( RI \) than a higher level page
\( p_0 \) if
\( p_0 \) is not an ancestor of
\( p \). Moreover, pages at
the same level can have different
\( RI \)s. Therefore,
neither depth-first nor breadth-first implementation
of the search procedure can guarantee this localized
search.

Although different index pages can have different
\( RI \) values, the proposed zoned-partitioning algo-
rithms guarantee that an index page always has
a lower \( RI \) value (i.e., a smaller area and margin)
than all of its ancestors at higher levels. That is,
along each search path downward the tree, the
involved zones are always in an ascending order by
\( zid \). Note that to guarantee that this condition
always holds, every modified page is written on the
disk by the function \textit{write_index_page} (Fig. 3) in a
single atomic transaction. Under this condition,
the proposed generalized search algorithm, \textit{Index_}
\textit{Search}, in Fig. 4 can ensure that each search is a
sequence of \textit{zone-bursts} and each involved zone is
associated with only one of the bursts of the search.
Furthermore, since the algorithm is generalized, it
can be applied to a vast majority of tree-type access
methods.

\textit{Index_Search} uses an array of queues, Index Page
Array \( IPA[NZ] \), to trace the search flow and to
guarantee the localized disk access (i.e., zone-burst).
Starting from the root, the algorithm visits pages
satisfying the given query condition. A page to be
accessed is queued in the array according to the
location of the page. For example, when a page is
located in zone 3, it will be queued in \( IPA[3] \). The
algorithm reads all pages in an \( IPA \) queue one by
one and an accessed page is dequeued. While a
page is being accessed, each index entry \( I \), (see
Section 2.2) in the page is evaluated and children of
the page that should be visited are queued in the
corresponding \( IPA \) queues. When the current queue
becomes empty, the algorithm moves to the next

\begin{algorithm}
\SetAlgoLined
\KwInput{a range query \( q \)}
\KwOutput{None}
\Begin{
/* Assume \( IPA[NZ] \) is an array of queues. 
\ie \( IPA[i] \) contains pages stored in zone \( i \) \( (0 \leq i \leq NZ-1) \). */

Add root page into a queue \( IPA[z (\text{root})] \);

\For {\( i=0; i < NZ; i++ \)} 
{\While {\( IPA[i] \) is not empty} 
{Get page \( p \) from \( IPA[i] \); 
\If {\( p \) is a leaf page} 
{Evaluate data entries of \( p \) against a query \( q \) and output results;}
\Else {\( p \) is a root or an intermediate page */
\For {each index entry \( I \) of \( p \)}
{\If {\( I \) satisfies conditions of \( q \)} 
{Add a child page \( p_I \) of \( p \) to a queue \( IPA[z (p_I)] \);}
end if
end for
end if
end while
end for
}
End
\}
\end{algorithm}

Fig. 4. Localized query processing algorithm, \textit{Index_Search}.
queue (i.e., next zone). This process is repeated until the algorithm reaches to the slowest zone (NZ – 1).

The following theorem formally ensures the localized access in an index search.

**Theorem 1.** The combination of Index Search and Write Index Page guarantees the zone-burst access.

**Proof.** We can prove it by showing that, for zone \( i \) and \( j \) \((i < j)\), there will be no page added to IPA\([i]\) while accessing pages in zone \( j \). Suppose that there is a page \( p_j \) in zone \( j \) and a child page of \( p_j \), say \( p_{cj} \). Then we can think of two situations:

1. If \( p_{cj} \) is stored earlier than \( p_j \), the Index Zoning algorithm allocates \( p_j \) in a faster zone than \( maxchildzone \). Thus, \( p_j \) will always be in a faster zone than \( p_{cj} \).
2. If \( p_j \) is stored earlier than \( p_{cj} \), the Index Zoning algorithm may allocate \( p_{cj} \) in a faster zone than \( p_j \). However, Write Index Page algorithm will successively reallocate \( p_j \) towards the root and \( p_j \) will be ultimately assigned to the same or faster zone than \( p_{cj} \).

Hence, by (1) and (2), \( zone(p_{cj}) \geq zone(p_j) \). Because no child of pages in zone \( j \) can be found in zone \( i \), there will be no page added to IPA\([i]\) while the Index Search algorithm accesses zone \( j \).

5. **Experimental results**

In order to validate that the proposed zoned-partitioning techniques and their localized query processing technique (LQP) can significantly improve the performance of access methods, we performed a set of experiments with the original R*-tree (with 30% forced-reinsertion at the leaf level) [12] and two versions of Zoned R*-trees called the Area-Zoned R*-tree and the Margin-Zoned R*-tree. In the original R*-tree, we assumed that index pages were placed on a disk without any consideration of physical zones (i.e., random placement). Both the static and dynamic update environments were used for the experiments. Without loss of generality, we call the zoned-partitioning algorithm with the static environment as Static-Zoning (as described in Sections 3.1–3.3) and the one with the dynamic environment as Dynamic-Zoning (in Sections 3.1 and 3.2). Static- and Dynamic-Zoning algorithms were implemented in both the Area-Zoned R*-tree and the Margin-Zoned R*-tree: while the Area-Zoned R*-tree used the area in calculating the RI of each individual index page, the RI of each page of the Margin-Zoned R*-tree was calculated based on its margin in the data space.

In the experiments, a disk model of Seagate Barracuda 7200.7 in Table 2 was used to calculate the page read/write time of each individual page access. The last column shows the average page read/write times in milliseconds when disk accesses are localized within a specific zone; assuming that disk access is limited within a specific zone for a certain amount of time, even though physical zone 0 provides the fastest data transfer rate, the effective page access time within it is not the smallest because of a longer seek time due to the fact that the zone consists of the largest number of cylinders. It is counter-intuitive that the smallest capacity zone provides the shortest effective page access time. This is because the seek time is a function of cylinder distance. For example, the average seek time within the physical zone 0 with 48 GB capacity is 1.47 ms while that within the physical zone 8 with just 6 GB is only 0.24 ms. Note that the average read/write time without localized accesses, i.e., conventional disk access model, is 11.67 ms.

### Table 2

<table>
<thead>
<tr>
<th>zid</th>
<th>Physical zone no.</th>
<th>ZCap (GB)</th>
<th>ZR</th>
<th>Average 8kB page read/write time in milliseconds (the zoned R*-tree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>6</td>
<td>0.03</td>
<td>4.41270302</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>6</td>
<td>0.03</td>
<td>4.45465719</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
<td>0.045</td>
<td>4.49172672</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>6</td>
<td>0.03</td>
<td>4.510334263</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>9</td>
<td>0.045</td>
<td>4.526061249</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>8</td>
<td>0.04</td>
<td>4.573418958</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>9</td>
<td>0.045</td>
<td>4.578541932</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>8</td>
<td>0.04</td>
<td>4.595950336</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>12</td>
<td>0.06</td>
<td>4.613985193</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>14</td>
<td>0.07</td>
<td>4.625634668</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>17</td>
<td>0.085</td>
<td>4.703298305</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>14</td>
<td>0.07</td>
<td>4.704762063</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>13</td>
<td>0.065</td>
<td>4.72599499</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>21</td>
<td>0.105</td>
<td>4.892135707</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>48</td>
<td>0.24</td>
<td>5.64322693</td>
</tr>
</tbody>
</table>

DCap (disk capacity), 200 GB; average seek time, 7.5 ms; average rotational latency, 4.17 ms; average data transfer rate of the disk, 50.47 MB/s.
The tested R*-trees were fully implemented, functioning access methods that physically read/write pages. The underlying Seagate Barracuda disk was simulated using the values in Table 2 and all the disk zones were used for storing the index structures (i.e., \( NZ = 15 \)). We assumed one-page-buffer and no-pre-fetching to concentrate our focus on disk I/O time.\(^2\) The page size was fixed at 8 kB, and all values (coordinate values and pointer values) stored in the tested structures were 4 bytes long.

5.1. Static-Zoning

In the first experiment with uniformly distributed data, the number of dimensions was varied between 2 and 64. The page size was fixed at 8K (8192 bytes). For each \( d \)-dimensional space \([0,1]^d\), a data file of 131,072 (\( 2^33 \)) random point objects was generated. Data objects (records) of each file were inserted into the R*-tree. Then the Margin-Zoned R*-tree and the Area-Zoned R*-tree structures were built by duplicating and zoned-partitioning the complete R*-tree structure. The retrieval performance of the access methods was measured over four sets of 2000 range queries. In first three sets, each side of a random query rectangle was obtained by generating a random center and extending it along the dimension parallel to the side by 0.02, 0.1, and 0.25. In the last query set, each query was generated as follows: (1) randomly generate the center point; (2) extend the query along every dimension so that the volume of the query is 0.0001. That is, in the last query set, each unclipped query range covered 0.01% of the data space. In all query sets, the query rectangles that intersect the boundary of the data space were properly clipped.

The results of the experiment given in Fig. 5a show that the zoned R*-trees improve the range query performance by reducing the average page access time (average time in seconds required for a query to read all necessary index pages from the disk). As shown in the figure, the Static-Zoning technique constantly reduces the average page access time per query compared to that of the R*-tree, and the Static-Zoned R*-trees are two to three times faster than the original R*-tree in processing range queries.

To further discuss the effectiveness of the proposed zoned-partitioning techniques, we defined the following measure: Average Percentage of Zoned Pages Accessed by a single query (APZPA). In both the Area-Zoned R*-tree and the Margin-Zoned R*-tree, we found that a faster zone (zone 1 is the fastest zone and zone 15 is the slowest zone in the graphs) is associated with a larger APZPA value (Fig. 5b). This means that the pages that were stored in a faster zone actually had a higher access frequency, and more pages required by a query were found in faster zones.

In the second experiment with 10 data clusters, each cluster was randomly located in the universe and had an extent that varied between 0.05 and 0.3 along each dimension. The clusters were populated using 101,072 random data objects. Then 30,000 points were randomly scattered through the entire universe. The results of the experiment given in Fig. 5c are similar to the results of the first experiments.

The last experiment was conducted over a large real data set. Our real data set represented a database table of 1,028,872 records. This data table was obtained from a database of a telecommunications company. The original table has 19 attributes of different types. However, by breaking the values of certain attributes and applying simple transformations, we obtained an array of 25-dimensional points with 4-byte unsigned long coordinates (nulls were replaced by a value of zero). Using an order-preserving domain transformation, the values of each attribute were normalized to a range of floating-point numbers between 0 and 1. Then, by multiplying each attribute value by \( 2^{32} \) and rounding the result, we obtained the data points in a universe with 25 dimensions, each of which was 4 bytes long. Analyzing this set, we found that exactly 303,278 objects had no null values and that each of the rest of the data objects (i.e., 725,594 records) had about 2.18 nulls on average. The objects having one or more nulls appear on the boundary of the universe. That is, over 70% of the data points were located on the boundary of the universe.

To measure the performance of the R*-tree variants, five sets of range queries were generated (see Fig. 5d). In the first two query sets, the side length (extent) of every query was fixed at 0.1 and

---

\(^2\)In transactional environments, many transactions and queries are concurrently processed over numerous data sets and indices, and each query is allocated only a small fraction of the database buffer. In addition, the disk-level scheduling and pre-fetching are not fully utilized because of the concurrent-transaction scheduling. Many database query optimizers (e.g., the cost-based query optimizer of Oracle) assume that each query will be executed on a busy multi-user system with a fairly low buffer cache hit rate.
As before, the queries that intersect the boundary of the data space were clipped. In the other three sets, query rectangles had a fixed volume of 0.0001, 0.01, and 0.1. Again, some queries were clipped. Every query set had 3000 queries.

The results of the experiment given in Fig. 5d also demonstrate the same trend as in the results of the previous experiments. However, in this experiment, the Margin-Zoned R*-tree further reduced the page access times of the Area-Zoned R*-tree by about 5–10%. One can find the reason for this by observing Fig. 5e. While the Margin-Zoned R*-tree's page accesses are nicely skewed toward the fastest zone, the Area-Zoned R*-tree shows more flat APZPA. That is, the Margin-Zoned R*-tree is better than the Area-Zoned R*-tree in estimating the access probabilities of the pages.

Fig. 5. Experimental results of the Static-Zoning.
difference was observed only in this last experiment because the skewedness of the real data set made the index rectangles have a much lower squaredness.

5.2. Dynamic-zoning

The experimental environment factors including the implementations of the techniques, data sets, and query sets were exactly the same as those of Section 5.1. However, to test the Dynamic-Zoning, data objects (records) of each file were dynamically inserted into all three variants of the R*-tree and the Margin-Zoned R*-tree and the Area-Zoned R*-tree structures were constructed and zoned on the fly as the data objects were being inserted.

The results of the experiment given in Figs. 6a, 7a, and 8a show that, like the Static-Zoning, the Dynamic-Zoning improves the range query performance by reducing the average page access time (average time required for a query to read all necessary index pages from the disk). As in the case of the Static-Zoning, the figures show that the proposed zoned-partitioning techniques (i.e., the Area-Zoned R*-tree and Margin-Zoned R*-tree) constantly reduce the average page access time per query compared to that of the R*-tree.

Unlike the Static-Zoning, the Dynamic-Zoning cannot guarantee that the selected zones are equally utilized at all times. To further discuss this aspect of the Dynamic-Zoning technique, we considered the zone utilization indicator (ZUI) defined in Eq. (2). Figs. 6b, 7b, and 8b show the ZUIs of both the Area-Zoned R*-tree and the Margin-Zoned R*-tree. Please note that, as defined in Eq. (2), a ZUI of 1.0 does not mean that the storage space of the zone is full, but means that a desirable amount of index data (i.e., the denominator in Eq. (2)) is stored in the zone.

For uniform data distribution, Fig. 6b shows that all zones are almost equally utilized especially in higher dimensional spaces. In low-dimensional spaces, zone 0 is underutilized. To understand the reason for this, one should consider the following properties of tree-type index structures with a fixed page size: (1) as the number of dimensions increases, the size of each individual entry becomes larger because each entry has more coordinate values; (2) as the number of dimensions increases, the maximum number of entries that can be stored in a single page decreases because the page size is fixed and the entry size increases.

These mean that the average fanout\(^3\) of a lower dimensional index tree is larger than that of a higher dimensional index tree. In addition, given the same number of data objects, a lower dimensional index

---

\(^3\)In a typical index tree that splits each overfilled page into two halves, the average fanout (page utilization) is about \(\ln 2 \times C\), where \(C\) is the maximum number of entries that can be stored in a page [12].

tree consists of a smaller number of pages than a higher dimensional index tree does. For example, in the experiment, the two-dimensional, 16-dimensional, and 64-dimensional R-trees consisted of 280, 1616, and 6550 pages. Therefore, storing one more or one less page in a zone resulted in a larger increase or decrease in the zone utilization in a lower dimensional case.

Moreover, the measure (i.e., area or margin) decreases downward the tree at a rate that is a function of the fanout. This rate of a lower dimensional structure is much higher than that of a higher dimensional structure, because of the larger fanout of the lower dimensional structure. This results in huge gaps among a few top $RI$ values. For example, in the experiments with two-dimensional
data set (i.e., two-dimensional case in Fig. 6b), zone 0 and zone 1 contained 2 and 9 pages, respectively, although both zones have the same capacity. That is, there was only zero (this is possible because of the page shifting of the Dynamic-Zoning algorithm in Section 3.2) or one page that had an RI value that was close enough to the root’s RI to position itself in zone 0.

In the ZUI results (Fig. 7b) of the skewed data, the Margin-Zoned R*-tree showed a higher ZUI on zone 0 when a high-dimensional data set was given. This is what we actually expected—given the same structure, the area of index pages decreases downward the tree at a much faster rate than the margin of index pages does, especially in a higher dimensional space. One can further observe this by comparing Definitions 1 and 2 in Section 3.1 with the changing factor d. This resulted in a couple of slightly underutilizes zones in slower, but larger, zones (zones 12, 13, and 14 in Fig. 7b). Consequently, preceding zones had too many pages and sent some pages towards zone 0. The ZUI results of the real data in Fig. 8b showed the best ZUIs that we have observed.

5.3. Discussion

The percentage improvements represent \(100 \times \frac{(RT_R^* - RT_{DIT})}{RT_R^*}\), where \(RT_R^*\) is the page access time (average time required for a query to read all necessary index pages from the disk) of the original R*-tree and \(RT_{DIT}\) is the page access time of the R*-tree improved by the zoned-partitioning technique (Area-Zoned or Margin-Zoned). In our comprehensive experiments, both the Static- and the Dynamic-Zoning constantly improved the performance of the R*-tree by about 60% in all tested cases.

As discussed in Section 1, when the pages that are stored in a faster zone actually have a higher access frequency (i.e., more pages required by a query are found in faster zones), the query performance is further improved. That is, we want a faster zone to have a higher \(APZPA\) value. From Figs. 5d and 8a, one can find that the improvements of the Margin-Zoned R*-tree for the real data are higher than those of the Area-Zoned R*-tree by a small margin although their zone utilizations are about the same. One can find the reason for this by observing Figs. 5e and 8c: the Margin-Zoned R*-tree’s \(APZPAs\) are better than those of the Area-Zoned R*-tree. While the Margin-Zoned R*-tree’s average page access frequencies are nicely skewed towards the fastest zone, the Area-Zoned R*-tree shows rather flat \(APZPAs\). That is, in our experiments with the real data, the Margin-Zoned R*-tree was better than the Area-Zoned R*-tree in estimating (i.e., RI, Eq. (3)) the actual access frequencies of the pages. On the other hand, although both of the Area-Zoned R*-tree and the Margin-Zoned R*-tree showed excellent (i.e., almost equal) zone utilizations in most cases, the Area-Zoned R*-tree showed a better ZUIs in the experiment with the synthetic skewed data (Fig. 7).

In closing this section, we put an emphasis on the fact that both the Static- and the Dynamic-Zoning can improve virtually any access method. In the paper, we presented two first attempts (one is the margin-based zoned-partitioning and the other is the area-based zoned-partitioning) that can enhance any access method that employs a hierarchy of pages as the index structure. The presented zoned-partitioning techniques can be directly applied to the following access methods without any modifications: KDB-tree variants [13,14], R-tree variants [11,12,15,16], QSF-tree variants [10], and B-tree variants [17]. Because rectangular approximations (or grouping) of regions are characterized by intermediate complexity and accuracy, they tend to be used more frequently. However, there are different access methods, such as the SS-tree [18], the TV-tree [19], and the Pyramid technique [20], that employ a different approximation of regions. One can easily apply the zoned-partitioning techniques to any hierarchical access method that employs a non-rectangular approximation or grouping by simply changing the measures in Definitions 1 and 2.

6. Conclusion

In recent years, an increasing number of database applications deal with large sets of static or dynamic multi-dimensional data. In these applications, the query performance is heavily dependent on the available access methods and the underlying disk system. Unlike traditional disk models, recently developed disk models provide multiple zones in a disk, where seek times and data transfer rates vary significantly across the zones. Although efficient index optimizations, such as bulk-loading and packing algorithms, have been developed for large static data, there is a marked lack of investigation
on how to optimize multi-dimensional access methods given a zoned disk model.

In this paper, we proposed a generalized zoned-partitioning technique that can significantly enhance the query performance of a wide variety of access methods by taking into account the different page read/write times of the disk zones. The proposed algorithm can partition virtually any static or dynamic hierarchical (tree-like) index structure in such a way that more frequently accessed pages are stored in a faster zone without skewed usage of zones. Then the proposed generalized query algorithm (localized query processing, or simply LQP) makes each search be a sequence of zone-bursts, where each involved zone is associated with only one of the bursts. In the presented experiments, both of the proposed zoned access techniques improved the query performance of the R*-tree by an almost constant factor of 2–3.

This paper assumes a single disk for a focused discussion. However, individual zones need not be originated from a single disk. In practice, they can be distributed across multiple disks in accordance with the underlying disk subsystem’s data placement policy. In the presence of intelligent storage systems that can logically construct virtual storage volumes (Logical Unit Numbers) of any size and service level, our partitions can be straightforwardly mapped to logical data block units across multiple disks. For example, 3PAR InServ storage server [5] provides a flexible data placement in aware of the performance difference between the inner and outer regions of a disk as well as the construction of a “virtual disk” based on uniform-sized storage units, called chunklets, across multiple disks. Then, the implementation of our algorithms on such systems can be defining a new set of data placement rules such as mapping our partitions to a set of chosen chunklets. This, in turn, can be used by the administrator or automated storage manager (e.g., Oracle 10g’s Automatic Storage Manager with Auto Extend) of an application for optimal storage utilization and query performance across multiple disks.

In our future research, we will consider zoned multi-disk models with concurrency control, and different type queries, such as the k-nearest neighbor query. In addition, defining appropriate zoning factors (the relative importance or RI) for individual pages of spatiotemporal access methods will be further investigated. Dynamic zoning of spatiotemporal access methods indexing continuously changing objects will be an interesting and challenging research issue because, due to the fact that the data objects can continuously change over time, the spatiotemporal access methods, such as the TB-tree [21], the 3DR-tree [22], and the TPR-tree [23,24], that are designed to support spatiotemporal queries referring to continuously changing data [25] are highly dynamic by nature.

Acknowledgment

This research was partially financially supported by Hansung University in the year of 2007.

References


