The VLDB Journal (2007) 16: 417–437
DOI 10.1007/s00778-005-0171-7

REGULAR PAPER

Christopher Jermaine · Edward Omiecinski · Wai Gen Yee

The partitioned exponential file for database storage management

Received: 18 June 2003 / Revised: 8 February 2005 / Published online: 26 July 2006

Abstract The rate of increase in hard disk storage capacity continues to outpace the rate of decrease in hard disk seek time. This trend implies that the value of a seek is increasing exponentially relative to the value of storage.

With this trend in mind, we introduce the *partitioned exponential file* (PE file) which is a generic storage manager that can be customized for many different types of data (e.g., numerical, spatial, or temporal). The PE file is intended for use in environments with intense update loads and concurrent, analytic queries. Such an environment may be found, for example, in long-running scientific applications which can produce petabytes of data. For example, the proposed Large Synoptic Survey Telescope [36] will produce 50–100 petabytes of observational, scientific data over its multi-year lifetime. This database will never be taken off-line, so bursty update loads of tens of terabytes per day must be handled concurrently with data analysis. In the PE file, data are organized as a series of on-disk sorts with a careful, global organization. Because the PE file relies heavily on sequential I/O, only a fraction of a disk seek is required for a typical record insertion or retrieval.

In addition to describing the PE file, we also detail a set of benchmarking experiments for *T1SM*, which is a PE file customized for use with multi-attribute data records ordered on a single numerical attribute. In our benchmarking, we implement and test many competing data organizations that can be used to index and store such data, such as the B+-Tree, the LSM-Tree, the Buffer Tree, the Stepped Merge Method, and the Y-Tree. As expected, no organization is the best over all benchmarks, but our experiments show that T1SM is the best choice in many situations, suggesting that it is the best overall. Specifically, T1SM performs exceptionally well in the case of a heavy query workload that must be handled concurrently with an intense insertion stream. Our experiments show that T1SM (and its close cousin, the T2SM storage manager for spatial data) can handle very heavy mixed workloads of this type, and still maintain acceptably small query latencies.

Keywords Storage management · Indexing · Data warehousing

C. Jermaine (✉)
Department of Computer and Information Sciences and Engineering, University of Florida, Gainsville, Florida, USA
E-mail: cjermain@cise.ufl.edu

E. Omiecinski
College of Computing, Georgia Institute of Technology, Atlanta, Georgia, USA
E-mail: edwardo@cc.gatech.edu

W. G. Yee
Computer Science Department, Illinois Institute of Technology, Chicago, Illinois, USA
E-mail: yee@iit.edu

1 Introduction

The majority of database access methods in use today are effectively based on the *I/O model* formalized by Aggarwal and Vitter [1]. In this model, disk-based data are organized into atomic units known as "blocks" or "pages," and the key to increasing the query processing and update rate is to reduce the number of pages read and written. However, hard disk performance trends over the past 30+ years force us to reexamine the validity of this model. In this paper, we explore an alternative data organization that takes advantage of notable trends in hard disk performance [7, 13], including:

- *Storage capacity* per disk has increased historically at a rate of 27% per year [7], doubling every 3 years. The rate of increase seems only to have increased recently. Now, mid- to high-end PCs are regularly being shipped with 200 + GB hard drives.
- *Seek time*, defined as the time required to move a disk arm plus rotational delay, has decreased only at a rate of 8% per year [7]. Modern seek times seem to have stalled at around 5 ms for a commodity disk.
- *The sequential data transfer* (I/O) rate has increased historically at a rate of 22% per year [7]. Although the sequential I/O rate has not kept pace with storage capacity (that is, the time required to sequentially scan an entire

disk has slowly crept upward), it has come much closer to keeping pace than has seek time.

We also point out that these are historical trends based on relatively old numbers [7]; an examination of recent hard disk specifications would seem to indicate that the rates of increase for storage capacity and sequential data transfer have accelerated. Consequently, a modern commodity disk can transfer data at a fast, sustained rate of 40+ MB per second. However, due to the constraint in seek time, the time required to complete a random I/O is not significantly less than it was 10 or 20 years ago. Furthermore, disk storage capacity has grown exponentially in size, increasing the disk's seek load per byte of storage. In response, the hard drive market has evolved, as now a variety of different drive characteristics are available (small or large capacity, slow or fast transfer rates, etc.). But the fact is that across the spectrum of modern hard drives, the number of seeks available per byte of storage is still orders-of-magnitude less than it was several decades ago. This is problematic, since common database storage structures such as the B+-Tree were developed at a time when the relative cost of a seek was far less than it is today.

In this paper we revisit the use of *exponential* (or *logarithmic*) file structures [5, 30] as a way of exploiting the hard disk performance trends described above. First proposed decades ago, exponential files are characterized by their sequential organization of data on disk; with it, fewer seeks are required to access data. They were largely ignored in production database systems, however, due to the reasonable performance of hierarchical, page-based access methods (e.g., the B+-Tree), as well as their own inability to scale with large datasets. Furthermore, seek conservation was not cost-effective in the past, especially considering the complexity introduced by previous exponential file implementations. The work in this paper demonstrates the potential performance advantage of exponential data organization on modern storage devices and describes the design of a simple and scalable storage manager based on it. Before proceeding, we introduce some concepts behind exponential file structures.

1.1 The exponential file

In exponential files, data are organized on disk as a series of ongoing merge sorts. Each sort, or *component*, is multiplicatively larger than the previous one. When a component fills with data, it is *flushed*, and its contents are *merged* (and sorted) with those of the next, larger component. For fast insertion, the smallest component can be pinned in memory. For fast data access within a component, each can be organized as a tree or hashed file. The basic idea is depicted in Fig. 1.

The promise of the exponential data organization stems from its basis on sequential disk I/O. Given the evolution of hard disk technology, there are two traits of this organization that are very attractive:

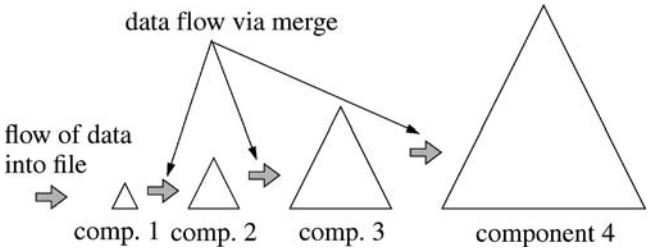

Fig. 1 Basic exponential file

- Updates depend almost exclusively on sequential I/O. Exponential files can therefore handle more intense update loads than page-based access structures.
- Data are written to disk in long runs. This allows us to organize data *among* disk pages in ways that minimize seeks during query processing. For example, we can tune inter-component organization in order to optimize the performance of range queries.

These benefits, however, are not without cost. There are two major reasons that the exponential data organization, as originally proposed [5, 30], is not practical for use in a real database system:

- The time required to merge two components is unbounded. This is a consequence of the unbounded component size. For example, merging two components that are GBs or TBs in size may take hours. This performance is debilitating in a database system where efficient, concurrent processing of queries is critical. As we will show, this is a factor in even the most modern variations of the exponential file paradigm, such as the *LSM-Tree* [29].
- Exact-match search time is a function of database size. Depending on when it was inserted, a given record can reside in any component. Searching for it therefore requires a search of every component in the worst case, negating the benefits of an exponential file over a page-based one.

In this paper, we present a novel data organization called the *partitioned exponential file template* (or *PE file* for short). The PE file is based on the exponential file, and can concurrently handle tremendous update loads and heavy, analytic query processing. The PE file divides data into many *partitions* (potentially thousands). Each partition corresponds to an exponential file of bounded size, but stores only records falling within a range of key values. The design is illustrated in Fig. 2.

The PE file alleviates the latency/exact-match problems to a great extent. Merge performance is improved because partition size, and therefore component size, is bounded. Exact-match query evaluation performance is improved because partition key ranges are disjoint, and therefore searches only span a partition.

The use of partitions has other benefits as well, including:

- A partition is a natural unit of concurrency control;

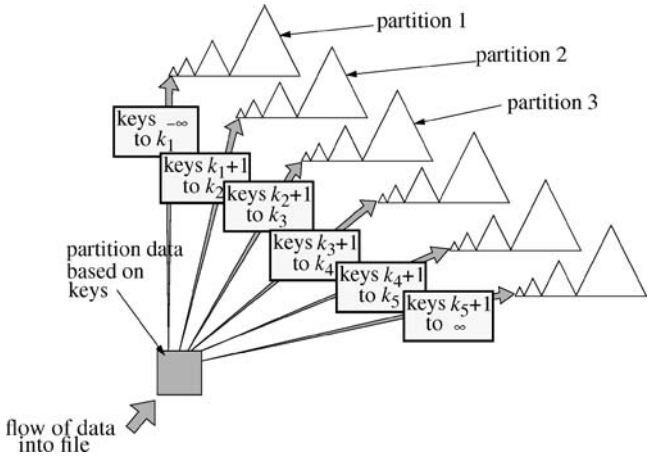

Fig. 2 Design of PE file

- Multiple partitions handle skewed update patterns gracefully, because buffer memory is concentrated with the partitions that are actively being updated.

Finally, the PE file can be treated as a *template* data organization. Like the generalized search trees of Hellerstein et al. [15], it is general enough to be adapted and optimized for many data types and data-intensive applications.

The major contributions of this paper are a detailed analysis of the PE file as well as comprehensive benchmarks on virtually every major data organization and access method suitable for data intensive applications. The analysis of the PE file includes a discussion on its design as well as the details of a particular PE file implementation called *T1SM*. By default, we assume in our discussion that the PE file stores multi-attribute records ordered on a single numerical attribute, unless otherwise specified.

The benchmarks include results from modern versions of the exponential file (i.e., the *LSM-Tree* [29] and the *Stepped Merge Method* [16]) as well as tree-based access methods, such as the classical B+-Tree and variants such as the *Buffer Tree* and the *Y-Tree* [3, 8, 17]. We also consider the option of not clustering the data on disk at all. That is, we could simply write the data to disk in the order they are received, and then maintain a secondary index on the data. To give the reader a preview of our results, in Sect. 5.4 we rank the various alternatives from first to last in each of six different categories (such as raw insertion processing speed, range query processing ability, and so on). While we stress that such rankings necessarily mask many of the strengths and weaknesses of each method, they still provide a reasonable summary of our results. We found that an instantiation of the PE file had the best average ranking (at 2.16), followed by the LSM-Tree (at 2.83) and the no-clustering option (at 3.33). Next was the Stepped Merge Method (at 3.83), followed by the B+-Tree (at 4.16), the Y-Tree (at 4.66), and the Buffer Tree (at 5.5). Thus, our benchmarks indicate the utility of the PE file for processing update-heavy workloads.

The remainder of the paper is organized as follows. In Sect. 2, we discuss the basic partition structure and the process of inserting data. Section 3 discusses some higher level details of the PE file organization. Section 4 describes the instantiation of a PE file template, called *T1SM*. Section 5 compares the performance of T1SM against other data organizations and access methods. Related work is discussed in Sect. 6. We conclude the paper in Sect. 7. In the Appendix, we discuss PE file concurrency control and recovery.

2 The PE file structure

In this section, we describe the structure and basic operation of the PE file. The PE file is composed of K data *partitions* and a *database engine*. Each partition, $P_i (1 \leq i \leq K)$, is defined by a range of key values, $(k_{i-1}, k_j]$, where $k_0 = -\infty$ and $k_K = \infty$. All partitions have a size limit, S_{part}, and total amount of data that the entire PE file can store is initially limited to S_{file} (See Fig. 2). The database engine is in charge of maintaining the partitions and handling user operations, e.g., data queries and insertions. We will describe partitions and PE file operations in more detail in the next sections.

2.1 Basic partition structure

Each partition is made of two distinct parts: the *header* and the *body*. We will discuss the body first. The body is composed of a series of M *levels*. Level i of partition j is denoted L_{ij} $(1 \leq i \leq M, 1 \leq j \leq K)$. (For simplicity, we will only use the second subscript indicating the partition when necessary.) In the simplest version of the PE file, L_1 for all partitions can store S_{lev} bytes (this requirement will be relaxed in Sect. 3.3). To achieve the best possible performance, the first level L_1 is stored in main memory; subsequent levels reside on disk. Each subsequent level $L_i (2 \leq i \leq M)$ is larger than its predecessor by a factor $F > 1$; i.e., level size grows exponentially. When a level overflows during insertion, its contents are flushed and merged with those of the next larger level. (Note that a level is analogous to a component of an exponential file, described in Sect. 1. The main difference between the two is that a level contains only data with keys that fall within its partition's key range, whereas a component may contain data with any key.)

The header serves as a *directory* for the partition, storing indexes for each level. A query begins by reading the header, which helps it guide its search through specific levels within a partition. This header is organized as a hierarchical, B+-Tree-style index. We will discuss such operations in more detail later in the paper.

2.1.1 Physical layout of a partition on disk

When a partition is created, a contiguous block of disk space is allocated for its header and the body. The header is allocated enough space at the beginning of the block to accommodate its maximum possible size, S_{head}. The body is pre-allocated enough space to completely accommodate the

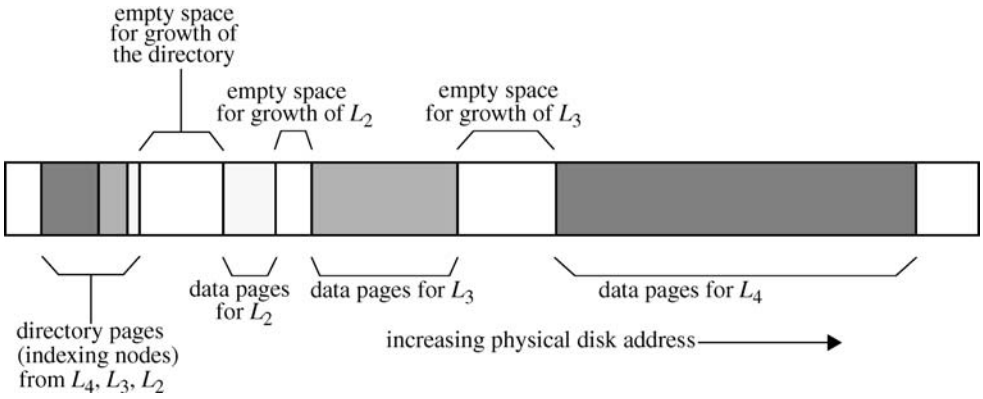

Fig. 3 Organization of a partition on disk

eventual growth of levels L_2 to L_{M-1}, i.e., $S_{\text{lev}} \times F \times (F^M - 1)/(F-1)$ bytes, followed by enough space to hold the data currently stored in L_M. The first portion of this space is allocated exclusively to level L_2, the next exclusively to L_3, and so on, and each portion is large enough to accommodate its level's maximum size. As a performance optimization, we pin L_1 of all partitions in memory, so it is not allocated any space in the body. Also, we only allocate enough disk space to hold the amount of data currently stored in level L_M. L_M consumes more space than all other levels combined, so deferring its full allocation increases flexibility in finding a suitably large extent on disk for the rest of the body. We will discuss handling changing space requirements for L_M later in this paper.

Level indexes are sorted in reverse order of the size of their respective levels, and packed to the left (low address) part of the header. Because smaller levels are more subject to change, so are their indexes. Placing them closer to free space in the header (the right part) makes updating them simpler. Increasing the size of the index for the smallest level, for example, can be done without perturbing the others. Packing indexes together allows multiple indexes to be read quickly. In general, the goal of header design is to allows its entire contents to be read with only short, forward seeks, which are more than an order of magnitude faster than random seeks (see Fig. 3).

In our prototype PE file, we index each level using a tree similar to a generalized search tree [15]. For simplicity, we will subsequently refer to this structure as a B+-tree, though like a generalized search tree it can be adapted for other types of data. Besides the design optimizations described above, each of this B+-tree's nodes are organized on disk in a breadth-first manner. Breadth-first organization ensures that traversing an index can be done using forward seeks (see Sect. 2.4 and Fig. 7). In addition, the first level of each index can be pinned in memory as a performance optimization. Additional index levels can be pinned in memory depending on the amount of memory available.

The data stored in each level are also packed together on disk as far left as possible, and conform to the organization implied by their respective indexes. In our case, the data are

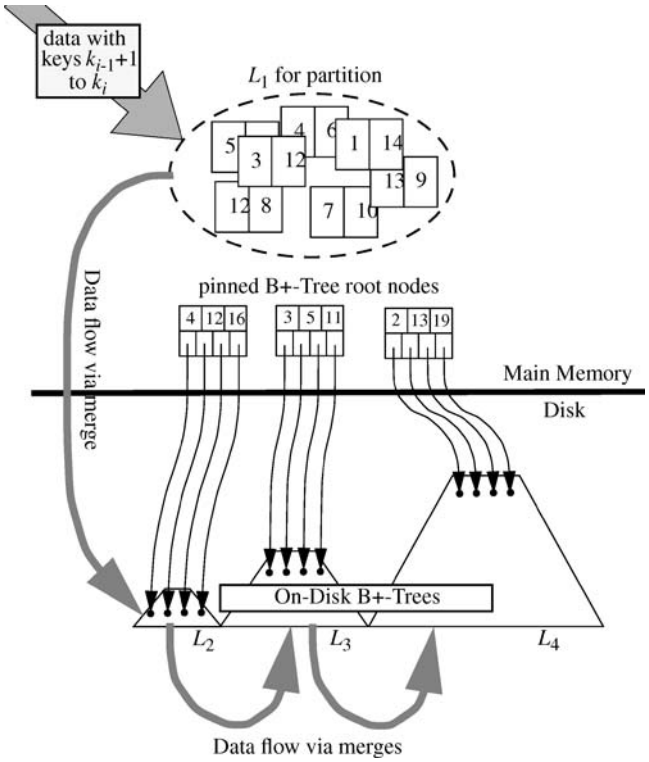

Fig. 4 Conceptual organization of a partition i

organized as the leaves of the B+-tree, and stored in key order within the disk space allocated for the level. The basic logical partition organization is shown in Fig. 4.

2.2 The PE file as a template: Handling other data types

As mentioned above, the PE file is a template that can be customized for use with various data types and data-intensive applications. It is for ease of presentation that we assume that data are identified by a one-dimensional numerical attribute and organized as a B+-tree. Although we could have based our discussion on other, tree-based organizations such as an R-Tree or a generalized search tree, doing so would

have increased complexity without adding any additional insight.

In the course of our discussion on the design of a PE file, some design decisions are data type specific. We will point out such cases, and discuss design alternatives. One technique we use to delineate and encapsulate application-dependent functionality is to introduce functions into the discussion, of the form *FunctionName*(*arguments*). This technique is meant to help both the researcher and programmer identify parts of the PE file that are meant to be customized per the application. We will treat application specific functionality in more detail in Sect. 4.

2.3 Insertion

Data inserted into a PE file are directed by the database engine to the partition based on each record's key value. Once a partition is determined, data are inserted into its smallest level, L_1. When a level is full, its contents are read from disk (if necessary) in a single sequential scan with the contents of the next, larger level. These data are *merged*, *sorted*, and *packed* (organized into pages and written to disk) into the larger level in a sequential write, leaving the smaller level empty. We call the merge, sort, pack operations *merge/pack* for brevity.

2.3.1 Example: Merge/pack of overflowing levels

Consider the insertion example shown in Fig. 5. Initially, items 4 and 8 are buffered in memory in L_1 (a). Inserting item 11 causes L_1 to overflow, so items 4, 8, and 11 are sorted and then written to L_2 on disk using a single sequential write (b). Items 2 and 15 are then inserted and buffered in memory in L_1 (c). Subsequently inserting item 16 causes L_1 to overflow. The contents of L_2 are then read from disk in a single sequential scan, then merge/packed with items from L_1, and rewritten to L_2 in a single sequential write (d). In (e) and (f), we insert items 7, 13, and 12 (in this order) to the partition. Item 12 causes L_1 to overflow, so we merge/pack L_1 with L_2. This process causes L_2 to overflow, so we merge/pack L_2 with L_3, yielding (f). In steps (*g*)–(*k*), we insert items 1, 5, 6, 3, 17, 18, 9, 10, and 14 into the partition. The last three items, 9, 10, and 14, cause L_1 and L_2 to overflow, forcing us to merge/pack levels L_1, L_2, and L_3. In this case, two levels, L_2 and L_3, are read from disk with a sequential scan during the merge/pack phase. When data are rewritten to disk with a sequential write, we get the state shown in (*k*). The process can easily be generalized to accommodate any number of levels.

2.3.2 Example: Effect of inserts on partition organization

An example of the effect of insertions on physical partition organization is given in Fig. 6. This figure illustrates how the partition layout addresses a fundamental goal of partition

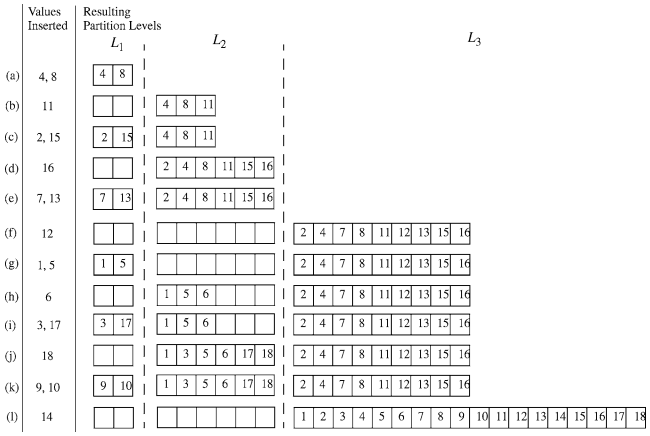

Fig. 5 Insertion into a partition

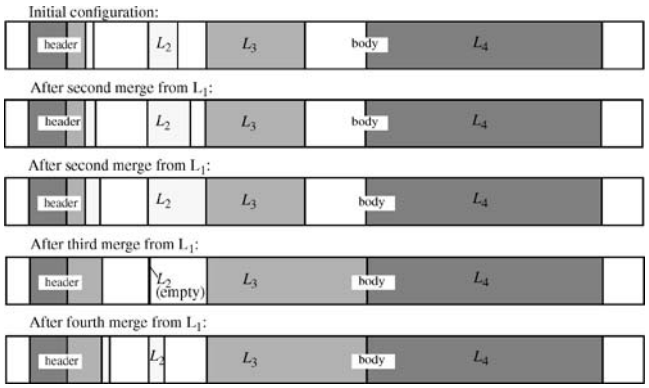

Fig. 6 Merges and their effect on partition layout

design: the minimization of I/O required to update the partition. This is achieved by ensuring that merges from L_1 into the on-disk partition levels have only relatively localized effects on the partition. In our example, L_1 is merge/packed four times with L_2. The first two merge/packs only affect L_1 and L_2. The third overflows L_2, forcing it to be merge/packed with L_3. L_2 is left empty, and the effect of the fourth merge is isolated to L_1 and L_2 again. Notice also the effect of merge/pack on the indexes in the header. The sizes of the indexes change in accordance with changes to the amount of data in the corresponding levels. Consequently, the index for L_2 is most dynamic. Ordering indexing in the header according to the inverse capacity of each level, and packing all indexes to the left minimizes the impact of the reorganization.

2.3.3 Overflowing a partition

Partitions are allowed to grow only to a certain size, S_{part}. When a merge into the highest level causes the partition to grow larger than S_{part}, the entire partition is read from disk in a single sequential scan, and its contents are split into multiple parts. The parts into which the partition are divided form the bases new partitions.

The nature of the partition splitting is application dependent and is encapsulated in a function called *SplitPartition*(P) where P is a partition (equivalently, the records in partition P). If the data are organized on a single, numerical attribute, it makes sense to split the overflowing partition in two, around the median element, similar to a B-tree split. But for other applications or data types (for example, the T2SM storage manager for spatial data [18]), a more complex or multi-way splitting scheme may be preferable. In general, the optimal number of new partitions and the initial sizes of each are subject to definition. Sect. 4 contains details on customizing a PE file for particular applications.

2.3.4 Insertion performance

At first glance, the merge/pack process seems to cause significant insertion overhead compared to more traditional data organizations. The opposite is true, however, for many reasons. First of all, partitions are assigned a maximum size, S_{part}, which is tuned to fit in memory. The limit on S_{part} bounds the amount of I/O and memory necessary to reorganize a partition, thereby limiting the overall reorganization cost.

Another reason PE file insertions are faster is its better use of buffer space. In a hierarchical file, a fundamental problem is that even given a huge amount of buffer memory, under a uniform insertion pattern, the probability that a random insert hits a buffered page is basically zero. For example, assume that the buffer is on the order of 1% of the database size. In this case, the probability that a random insert hits a buffered page is also 1%. The probability of a buffer miss and the resultant disk access is therefore 99%.

Alternatively, one may use buffer memory to cache inserts instead of leaf pages, flushing the inserts to disk in a batch. If multiple insertions are destined to be inserted into the same leaf page, then they may be processed at once in this case [32]. Performance, however, would not be much better. Again, assume that buffer memory is 1% of the database size, leaf pages are 16 KB, and the insertion pattern is uniform. In this case, when the buffer is full, the amount of buffer data that corresponds to an average leaf page is only 1% of 16 KB, or about 160 B. In other words, only a few records will be flushed to an average leaf page when the buffer is full. Even if the entire access tree (except the leaves) is pinned in memory, accessing a leaf would still require two seeks per insertion: one to read the leaf page and one to write it. For an intense insertion load, those two seeks can be debilitating.

In a PE file, because partitions are orders of magnitude larger than leaf pages in a hierarchical structure, we need to support fewer of them, and can therefore maintain larger buffers for each (in practice, on the order of 100 KB). These larger buffers allow us to buffer more inserts, dramatically reducing the number of seeks required during update processing. We concede that if pages in a hierarchical structure were stored contiguously on disk, then we could read these pages in sequence in order to insert more records per seek in a way similar to the PE file. However, we must then handle the difficult issues associated with keeping those pages contiguous as different regions of the data grow through insertion. This is a non-trivial issue, and has been the focus of considerable research [28, 29, 31, 37]. We will consider some of these other options experimentally in Sect. 5.

We also point out that dividing a partition into multiple levels mitigates the higher transfer costs associated with reading and writing large amounts of contiguous disk space by reducing the number of bytes transferred. The total number of bytes read/written can actually be *lower* in a PE file than in a hierarchical file. See Appendix A for an analytical treatment of these issues.

2.4 Queries

The database engine first determines the partition(s) that could contain data that match the query, based on their key ranges (See Fig. 2). For each of these partitions, a query evaluation thread is created that checks L_1 (the level pinned in memory) for matching data. Each thread then obtains exclusive access to the disk containing its partition, protecting the access with a semaphore. Such access is necessary to avoid competing queries from moving the disk arm to other partitions. The thread then proceeds with searching the portion of the partition stored on disk. It first performs a possibly long disk seek to the beginning of the partition, and reads the directory information (the indexes) contained in the header, as depicted in Fig. 7. Depending on the directory, the pages from levels are read in the order they appear on disk: first, the relevant L_2 pages are read in sequence, followed by the relevant L_3 pages, and so on.

2.4.1 Query performance

The nature and limited number of seeks are the sources of the PE file's query evaluation speed. Because partition levels are contiguously stored on disk, usually only fast, track-to-track seeks are needed so scan them. Furthermore, only a limited number of seeks is required to search a partition: one seek is required to reach the header for the partition, and L-1 seeks are then required to retrieve the relevant pages from each of the L-1 levels stored on disk. Because blocks are packed tightly within each level, this is the case even for

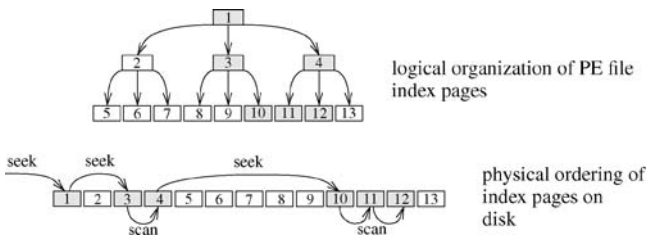

Fig. 7 Linearization of index directory pages in the header of a partition. Note that a range query that needs to touch index leaves 1, 3, 4, 10, 11, and 12 requires only a few short, forward seeks

larger range queries (as long as they stay within a partition). Most I/Os are fast, sequential reads.

2.4.2 Long-lived queries

Although the obvious benefit of locking the disk in this manner is that reads are accomplished in an orderly fashion with little thrashing, the drawback of such locking is that concurrent writes (as well as other query threads) will not be granted access to the disk while a query is reading a particular partition. If the query must read the entire partition, this could cause all subsequent disk operations to block for several seconds as the partition is read.

In practice, this problem will be mitigated to a certain extent for insertions because L_1 is pinned in memory, so subsequent insertions can be processed as long as the memory available to buffer L_1 has not been exhausted. If the available buffer memory is exhausted during a particularly long-lived query, then the simple solution is to block subsequent insertions until the query finishes. Sections. 5.3.3 and 5.3.4 give experimental evidence that even in an environment characterized by bursts of heavy insertions with concurrent queries, this simple solution has acceptable performance characteristics.

However, there will be certain application domains where the serialization forced by locking of the disk in this fashion is simply unacceptable. One example is online aggregation, where the answer to the query is computed incrementally, and a current estimate for the eventual answer is made available to the user at all times, with associated statistical error bars. In online aggregation, it would not be acceptable to access the partitions serially, because no statistical information would be available for the partitions that had not been processed. This would effectively lead to infinitely large error bars being associated with the online estimate. In this particular case (and for other applications with similar characteristics), it would probably be necessary to allow some degree of parallel access to the partitions relevant for answering a query, and to accept the inefficiencies incurred as the result of a less-controlled usage of the disk arm.

2.5 Deletions

The manner of handling deletions is similar to that of insertions, and are not described in detail. Basically, when a tuple is to be deleted, a *control record* containing its key is *inserted* into a partition. However, a flag on the control record is set indicating that the data record with this key is to be deleted. When the control and data record are merge/packed into the same partition, both are deleted.

2.5.1 Deletion performance

Storing a control record until it reaches its targets may incur some overhead, but because it only consists of a key value, the expected overhead is small. If overhead becomes a problem, the partition can be rebuilt to remove deletion records.

3 Details of the PE file data organization

3.1 Main memory usage

By definition, the smallest partition level L_1 is memory-resident. In practice, L_1 is on the order of 0.1–1% of the size of the overall partition, S_{part}. This proportion is governed in part by the ratio of the cost of main memory to the cost of magnetic disk per unit of storage. Historically, this ratio has hovered around 50:1 [12], which means that buying main memory for L_1 increases storage cost by about 2.5% (L_1 is 0.1% of S_{part}) to 25% (L_1 is 1% of S_{part}), assuming that the disk-based structures are mirrored for fault tolerance. This additional cost is acceptable in most situations, especially considering the performance increase, which we will demonstrate later.

3.2 High-level maintenance of a PE file

For a large database, thousands of partitions may be stored on disk. Partitions can be of many different sizes, with sizes differing by orders of magnitude. This presents a problem because, as partitions are constantly growing and shrinking with updates, the PE file can become extremely fragmented. For example, consider Fig. 8. Because of a merge into its highest level, partition 6 (which was formerly located between partitions 4 and 5) has grown too large to fit in any empty, contiguous storage in the PE file (the unpatterned spaces in the figure), even though there is clearly enough total room to accommodate it.

To reclaim large extents of space, the PE file must be reorganized by *moving* other partitions. Moving a partition consists of first identifying empty contiguous chunk of disk space and a partition currently on disk that would fit into it. The entire partition is then read into memory, written into the identified space, and the old space it occupied is marked as empty. As an optimization, the contents of a moved partition are merge/packed into its highest level while in memory. In this way, we take advantage of the I/Os already taking place by eagerly performing some in-memory reorganization of the partition.

Reorganizing a PE file is a complex task, but the final plan should minimize the following expression:

$$\sum_{P \in \{\text{partitions moved during reorganization}\}} \text{Cost}(P)$$

In the expression above, Cost(P) is the cost of moving partition P, and its exact definition is application-dependent. For example, moving a partition containing data organized on several attributes may require more time and CPU resources than moving a partition containing data organized on a single attribute. One reasonable cost function for simple, single-attribute data is Cost(P) = (size of P, in

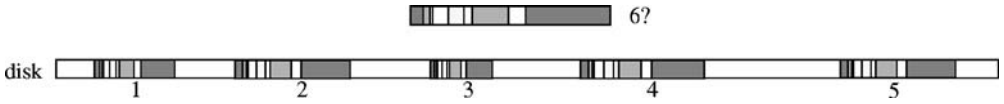

Fig. 8 PE file fragmentation

bytes). That is, the larger a partition, the more costly it is to read and write it again. A more complicated cost function is Cost(P) = (size of P, in bytes) × (% empty space in levels L_1 through L_{M-1}). In this case, cost is a function of the volume of empty space left in the partition, which roughly reflects the inverse probability that P will soon be merge/packed. By moving P, we can merge/pack P and reclaim disk space at the same time.

Even using a very simple cost, computing an optimal reorganization plan is an NP-hard problem (using a reduction from *bin-packing*). To deal with this complexity, we define the following classes of reorganization:

Definition 1 A **level** 0 **reorganization** to accommodate a partition P is a reorganization where there exists enough empty, contiguous storage on disk to accommodate P.

Definition 2 A **level** r **reorganization** to accommodate a partition P is a reorganization where for each partition P' that overlaps the contiguous storage where P is to be moved, P' can itself be accommodated by a level $r-1$ reorganization.

In the definitions above, r is a user-defined value and governs the maximum complexity of reorganization. In our experience, the optimal level 1 reorganization is easy to implement, is quickly computed using brute-force, and yields good performance. We have observed an 80–90% space utilization in a PE file using optimal level 1 reorganization in our prototype.

The general algorithm for maintenance of the PE file is as follows. We start with a small, arbitrary initial size for the PE file, $S_{\text{file}} \geq cS_{\text{part}}$, for some small $c>1$. Data are inserted, and whenever any partition grows because of a merge into L_M, we perform the optimal, level r reorganization of the file, based on the definition of Cost(). If no such level r organization exists, the PE file is declared full. In this case, we allocate more disk space to the PE file by increasing S_{file} by a user-defined factor *exp_fac*, and try the level r organization again. In our prototype, we set *exp_fac* to 1.1; when a PE file with $S_{\text{file}} = 1.0$ GB is full, it grows to 1.1 GB.

3.3 Accommodating skewed insertion patterns

Another issue that must be discussed in the design of the PE file is a method for dealing with skewed insertion patterns. In practice, certain partitions at certain times receive the bulk of the workload. The PE file described so far does not handle such workloads well. Say we have a 1 TB database organized into 20,000 50 MB partitions using a total of 10 GB of buffer memory. If we uniformly allocate buffer memory across partitions, each receives 500 KB of RAM for its memory resident L_1 level. In this case, if only a few partitions are being updated, the buffer memory for the partitions that are not accessed effectively go unused, wasting hundreds of megabytes of memory. This buffer space would experience higher utilization if it were allocated to the partitions that are actively receiving updates.

To deal with this problem, we modify the basic insertion algorithm depicted in Fig. 5. All available main memory (in our example, 10 GB) is organized into small pages and put together to form what is called the L_1 *buffer pool*. This buffer pool is kept separate from the normal pool of DBMS buffer pages. When a partition runs out of space in its L_1 buffer, it is allowed to draw another page from the L_1 buffer pool to add to its own L_1 buffer.

When the L_1 buffer pool holds no additional free pages, we invoke the *ChoosePartitionToMergePack()* function. This function chooses a victim partition that is to have its L_1 merge/packed with its upper levels (as in Fig. 5). After merge/packing the victim partition surrenders its L_1 buffer pages, which are empty, to the L_1 buffer pool.

Just as with the *Cost()* and *SplitPartition()* functions, the definition of the *ChoosePartitionToMergePack()* function is application-dependent. The most obvious solution is to choose the partition with the largest L_1; since this will immediately free up the most buffer pages. However, we have found in our prototype that this can be a poor choice in practice, since common insertion patterns can easily defeat this scheme. For example, imagine that key values were added *almost* in monotonically increasing order, but that some "older" key values were periodically inserted out of order. For example, consider temporally clustered data, where the data are generally inserted in the order that they are produced. If some data items are delayed or arrive out of order (due to network latencies, for example), one would frequently expect such an insertion pattern. The problem in this case is that data objects which are inserted out of order will be buffered in memory associated with partitions having older or smaller key values, but the partition holding the newest or greatest key values will always be the one that has its L_1 buffer pages returned to the L_1 buffer pool. Thus, the data objects inserted out of order are essentially buffered indefinitely in memory, when they should have been written to disk long ago. Due to this phenomenon, in our prototype, we choose to empty L_1 from the *least-recently-added-to* partition 20% of the time, and we empty the partition with the largest L_1 80% of the time. Choosing the partition with the largest L_1 most of the time makes sense, since it frees up the most L_1 buffer pages. On the other hand, choosing the least active partition periodically alleviates the problem of having costly memory associated with infrequently updated partitions. Although this 80/20 split is admittedly somewhat arbitrary, one of the benefits of the PE File is that it is customizable (see Sect. 4). Because it is a general-purpose

template, other schemes can easily be used, such as the LRU k algorithms described in [27]. When combined with a periodic emptying of the largest L_1, an LRU k algorithm may do an even better job of identifying those partitions which are unlikely to need buffer memory associated with them in the near future.

4 Instantiating the PE file

To tailor a PE file to a specific application (or data type), certain functionality must be defined, in a manner similar to the generalized search trees of Hellerstein et al. [15]. Below is a list of the major PE file functions and a description of their actions that are not common to both the PE file and generalized search trees. Altering their definitions allows substantial customization of the framework. In addition to the functions described below, several functions/parameters are common both to the PE file and to generalized search trees (specifically, these functions are *IsOrdered()*, *Compare()*, *Consistent()*, *Union()*, *Compress()*, and *Decompress()*). Because these are not substantially different than in a generalized search tree, we refer the reader to Hellerstein et al.'s paper for descriptions of these functions. The remainder are described below:

- *Pack(array of data records)*. When data from a lower level of a partition are merge/packed with data from a higher level, two basic operations must be performed in addition to basic disk operations. First, all of the data in the higher level must be organized into page-sized partitions. Second, data within the partitions must be ordered to reduce seeks during query evaluation. These operations are encapsulated in the *Pack()* function.

For example, if data are organized on a single attribute, *Pack()* is simple: it sorts the data records based on the key attribute, and insert page breaks when appropriate. With spatial data, however, *Pack()* is slightly more complex. In Fig. 9, we show one way to pack spatial data shown in (a). First, the data from (a) are packed into pages using the STR packing algorithm [21], as shown in (b). These pages are then ordered using a Hilbert curve (see Kamal [19]) (c) and written to disk (d). We implemented this packing technique in T2SM. T2SM [18] is an instantiation of the PE file that is suitable as an alternative to the R-Tree [14] or TSB-Tree [24] for spatial or multidimensional temporal data.

- *SplitPartition(array of data records)*. As described in Sect. 2.3.3, over-full partitions must be split. The *SplitPartition()* function splits the data from the over-full partition two or more parts, based on the application. Multiway splits can greatly improve global organization when compared to a hierarchical file. For example, for spatial data, data records can be split into many parts to mimic the natural clustering structure of the underlying data, whereas a traditional two-way split, common for data organized on a single attribute, might force an unnatural organization on the data (Fig. 10a).

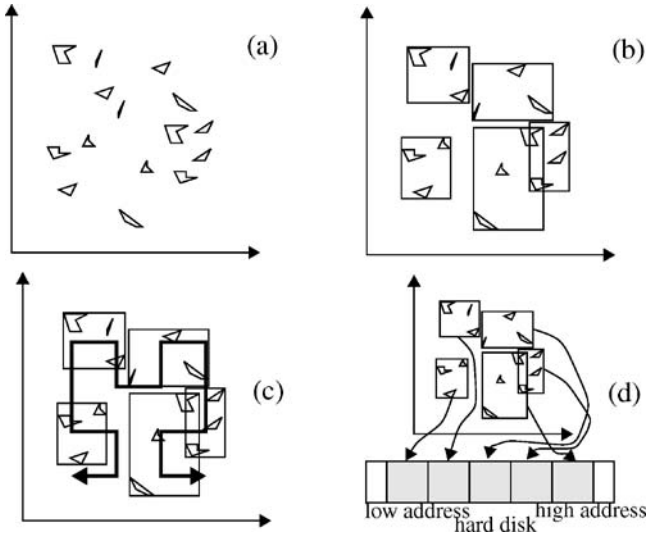

Fig. 9 Steps for packing a partition over spatial data

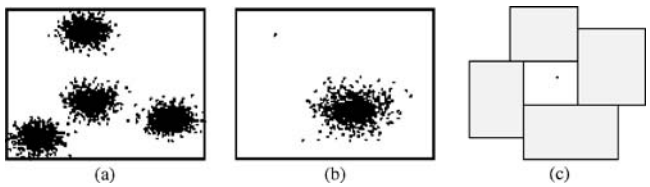

Fig. 10 Examples of spatial data

- *ChoosePartitionForInsertion(data record)*. At first glance, this may seem similar to the task of choosing which subtree receives an insertion in a hierarchical file. However, a key additional degree of freedom is provided in a PE file. *ChoosePartitionForInsertion()* may decide that no partition is appropriate for the new record, and create a new partition for it. There are many cases when such an option is useful. Consider the cases where the new record is an outlier, (Fig. 10b), or where expanding a partition to include the new record would cause significant overlap among partitions (Fig. 10c). In both cases, it may be preferable to create a new partition holding a single data record. This option is possible because there is no restriction on the minimum size of a partition; no analogous option is provided in a hierarchical file. *ChoosePartitionForInsertion()* can also be made to handle special characteristics of the input data, such as large runs of duplicate key values. If the case ever arose where a split caused two partitions to cover the same key value (one had values k_1 to k_2, the other had values k_2 to k_3), then the PE file could be made to use the heuristic that whenever a new record with key k_2 is added to the file, the record is added to the smaller of the two relevant partitions. If the file had even more partitions covering the same range of key values, a similar heuristic could be used.
- *Cost(partition)* and *ChoosePartitionToMergePack()*. These are more systems-oriented functions that were already described in Sects. 3.2 and 3.3, respectively.

In addition, the following parameters must be set. All have been described previously:

- S_{file}, the maximum size of the PE file. Discussed in Sect. 3.2.
- Main memory buffer size. Discussed in Sect. 3.1.
- F, the multiplicative factor describing the size difference between consecutive partition levels. See discussion for M, below.
- M, the number of levels in a partition. The choice of M determines the value of F. Increasing M increases insertion speed, but decreases query evaluation speed. Each increment of M increases the number of seeks within a partition during query evaluation by one.
- r, the maximum level of reorganization. Discussed in Sect. 3.2.
- *exp_fac*, the expansion factor of the PE file. Discussed in Sect. 3.2.

5 Benchmarking

We now consider the performance of the PE file. In the majority of our experiments, we benchmark the performance of the *Taboose One Storage Manager* (T1SM), which is an instantiation of the PE file for use with numerical data organized on a single attribute. We compare T1SM with six alternative data organizations that represent the state of the art for processing a workload of intense, concurrent queries and insertions: B+-Tree, Y-Tree, Log-Structured Merge Tree, Stepped-Merge Method, and No Clustering (all described below). We have also conducted experiments on spatial data, comparing the performance of the Taboose Two Storage Manager (T2SM), which is a PE file instantiated for spatial data, with an R-Tree in processing concurrent insertions and queries. For brevity, Sect. 5.7.4 only contains a subset of our spatial data results. A more detailed description of T2SM, including benchmarks, can be found in [18].

5.1 Data sets

Our experiments consist of several different insertion and query workloads over two synthetic, one-dimensional datasets called *Skewed* and *LessSkewed*. Both *Skewed* and *LessSkewed* consist of 256 byte records, organized over 4 byte integer keys having values from 0 to 2^{31}. Each set contains 39.06 million records, which amounts to 10 GB of total data. The keys for records of the *Skewed* dataset are produced by a one-dimensional, normally distributed cluster which slowly wanders through the data space. This cluster provides a single "hot spot" for insertion. The standard deviation of the cluster is such that around 5–10% of the data space could be expected to be actively receiving insertions at any given time. The *LessSkewed* dataset is produced in a similar way, except that it has 200 such clusters. While data insertion in *LessSkewed* is considerably more diffuse than in *Skewed*, its insertion pattern is still far from uniform. Because insertion into a real database is unlikely to be uniform over time, these two datasets represent fairly realistic insertion workloads.

5.2 Benchmarking environment

All experiments are run on a single processor, 1.2 GHz Pentium PC with 512 MB of RAM, running the Linux kernel, version 2.2.9–19 mdk. Each of the data organizations is implemented in C++. On-disk data structures are built on a 40 GB hard disk, managed using the Linux ext2 file system. Each data organization interfaces with ext2 through the Linux kernel via our own virtual file system, which provides additional facilities such as large (multi-gigabyte) files, LRU read/write caching, pinning of disk blocks, and pre-allocation of space for the large files (forcing them to be stored sequentially on disk). Each data organization is allowed 100 MB of main memory for use in storing the persistent main-memory data structures required by the organization (such as the memory required to buffer L_1 in the case of the PE file). If the data organization does not require any such data structures (for example, a B+-Tree will not require any long-lived structures other than the pinning of the root node), then that 100 MB is managed by our virtual file system as an LRU read/write buffer. This provides for the fairness of the comparison, since every data organization has 100 MB to use for long-term storage. Each data organization is allowed to use as much memory as is needed for short-lived operations (for example, many of the data organizations end up sorting large, in-memory sets of records), but any memory in excess of the 100 MB can only be used temporarily by the data organization.

The primary reason for managing the raw disk using the ext2 file system via the Linux kernel (and not directly using our virtual file system) was to provide a simple way to allow all of the system memory not used by each data organization to be used as a file system cache, without having to implement our own memory management software. In other words, all processes associated with a data organization request whatever memory they need via calls to *malloc()*, and we leave it to the operating system to handle issues such as removing memory from the file system cache to service the request, providing protection by making sure that all memory lookups remain in the correct address space, and so on. While a modern DBMS will usually handle these tasks internally, in our experience the Linux kernel does a good job of handling these tasks in the absence of other, concurrently running applications. Thus, we believe that this represents a reasonable environment for our benchmarking. We do not consider concurrency control in the initial set of experiments, but do consider it in Sect. 5.7.3.

5.3 Data organizations tested

The various parameters given below for each of the different organizations are chosen to maximize the performance

of each of the different options, as dictated by our experience with implementing and using the different organizations. The seven organizations that we tested are:

- T1SM. This is the instantiation of the PE file for use with one-dimensional data. Maximum partition size S_{part}= 12.5 MB. The number of levels in each partition $M = 3$. The amount of memory available for the L_1 buffer pool is held constant at 1% of the total size of the data that had been loaded into the structure. Because T1SM (and all of the other organizations) is allowed to use 100 MB of memory, the memory not used in the L_1 buffer pool at any given moment is used as a separate LRU buffer for disk reads/writes. The maximum level of reorganization $r = 1$. The expansion factor of the PE file, $exp\_fac = 1.1$. Disk page size is 8 KB.
- B+-Tree. We use a classical B+-Tree as a baseline for comparison. B+-Tree pages are 8 KB in size. The B+-Tree uses a 100 MB LRU buffer.
- Y-Tree [17]. The Y-Tree is similar to the B+-Tree, except that it is designed specifically to allow fast insert processing. All pages in the Y-Tree tested are 98 KB in size, and the insertion set size is 54 tuples. A 100 MB LRU buffer is used in conjunction with the Y-Tree.
- Buffer Tree [3]. The Buffer Tree is another tree-based structure, with design goals similar to those of the Y-Tree. However, the Buffer Tree is more suitable for use as a primary file organization holding larger records than the Y-Tree. We use 8 KB-sized pages in the Buffer Tree. As described in the original Buffer Tree design, 8 KB of data are inserted at a time, and tree nodes can store 4 MB of data. It is assumed that there is enough memory to store an entire tree node. This node size strikes a balance between insertion and deletion performance. Larger tree nodes improve insertion performance, but hurt query performance, as more data must be read and processed to satisfy searches. Finally, we assume that queries must be evaluated immediately, so our version of the Buffer Tree does not allow *lazy query evaluation*, as originally prescribed by its design [3]. All queries actively traverse the tree until satisfied. Again, a 100 MB LRU buffer is available to the Buffer Tree.
- The Log-Structured Merge (LSM) Tree [29]. The LSM-Tree is essentially a modern variation on the exponential file organization described in Sect. 1. We use a three-component LSM-Tree. The smallest component is 100 MB in size, residing in main memory. The second component is given 1 GB of disk space, and the largest component is given 10 GB of space. We use multi-page runs of 64 pages [29].
- The Stepped Merge Method [16]. The Stepped Merge Method (SMM) is best viewed as a variation of the LSM-Tree, allowing multiple trees at each level. This allows us to avoid writing any record more than once to a level (in contrast to the LSM-Tree). As it was originally defined, the SMM writes data to levels in a set of monolithic merges (or reorganizations). This is problematic while processing loads of concurrent queries and insertions (like the loads we test experimentally), because of the latency caused by these reorganizations. To address this problem, we use a variation of the *rolling merge* technique of the LSM-Tree [29] to perform merges in the SMM. Multi-page runs of 96 pages are used. We use twin 50 MB AVL trees (Gormen et al. [9], Chapter 6) for the lowest level. One is filled with new data, as the other is emptied into the next level. Then, there are four additional levels of up to four components each.
- No Clustering (NoClus). The last organization we consider is to simply write the data to disk in the order they are inserted (which will allow very fast insertion processing), and to then maintain a secondary index on the data to facilitate fast lookups of individual records. We use a Y-Tree for the secondary index, with 16 KB pages, as the Y-Tree supports very fast insertions for a secondary index where records are small. Again, a 100 MB buffer is used.

5.4 Organization of the remainder of Sect. 5

Using the *Skewed* and *LessSkewed* datasets and the six data organizations described above, our benchmarking is organized into three distinct sets of experiments. In the first set of experiments (described in Sects. 5.5 and 5.6), we test the standalone insertion and query processing speeds of each data organization. To facilitate this testing, each organization is used to load up each dataset as fast as possible. To test query speed, the loading process is periodically halted to issue non-noncurrent test queries. Insertion rates are discussed in Sect. 5.5, and query processing speeds are discussed in Sect. 5.6.

We also describe two sets of experiments designed to test the ability of the various data organizations to handle concurrent queries and insertions. These results are reported in Sect. 5.7. First, we describe a set of experiments that are aimed at determining the maximum query load that can be processed along with a given insertion load, for each of the six data organizations. These experiments are discussed in Sect. 5.7.1. In a final set of experiments (described in Sect. 5.7.2 and 5.7.3), we test the efficacy of the concurrency control mechanisms associated with the PE file. In particular, these experiments explore the effect of concurrent insertions on query latencies. The PE file is tested as a data organization both for one-dimensional and for spatial data, and its concurrency control mechanisms are compared head-to-head against the concurrency provided by both a B+-Tree and an R-Tree. Finally, Sect. 5.8 summarizes our overall experimental findings.

5.5 Insertion

5.5.1 Experiments

In our first set of experiments, we measure the amount of data each access structure can insert in a fixed amount of

time. The size of the dataset is 10 GB and the time limit is 12 h. For these experiments, we measure average and worst-case insertion performance.

5.5.2 Average (or overall) insertion time

We can draw some interesting conclusions from the experiments. First, simply writing data to disk in the order they arrive, and maintaining a secondary index with a fast structure such as a Y-Tree (e.g., using NoClus) is likely to be the fastest method of processing insertions, especially if most of the index can fit in memory. The size of the secondary index required is on the order of 500 MB for each of our two datasets. In the *Skewed* dataset, LRU buffering of index disk pages is more efficient and the active portion of the index could fit in memory, explaining the faster insertion speed for NoClus over the *Skewed* dataset, as opposed to *LessSkewed*. The SMM is also very fast, since with the configuration we use guarantees that no single record item is written more than four separate times to disk. The SMM insertion processing speed (and also the speed of the LSM-Tree, with which it shares many characteristics) is nearly independent of data distribution. Note that T1SM is competitive with the other methods tested. The results of this experiment are shown in Fig. 11.

5.5.3 Worst-case insertion time

Overall insertion time only tells part of the performance story. With the exception of the B+-Tree, each of the data organizations considered works to some extent by processing updates in batch. This leads to fast *amortized* insertion times, but may mean that *individual* updates can take a long time to process. Long individual insertion times are problematic because in each of the structures tested, long-lived insertions are effectively *blocking*. Due to problems with disk arm thrashing, loss of sequential I/O, and the blocking of additional insertions, long-lived insertion make efficient co-evaluation of queries difficult (Table 1).

Most of the organizations have acceptable worst-case insertion times, except for the Buffer Tree. The phenomenon of long-lived buffer tree insertions is related to the fact that when a Buffer Tree node's associated buffers are full, the buffers' contents are emptied in a series of flushes into all its children that have the corresponding key ranges. The more uniform the insertion pattern, the more children are affected, resulting in more I/O. Furthermore, this emptying process is recursive. If the majority of the children of a node are almost full, then a single emptying into all of a node's children can cause all of those nodes to overflow as well, which can result in cascading reorganizations that affect most of the Buffer Tree. The problem is worse with more uniform insertion patterns, where there is a high likelihood that all children of a node tend to fill up at around the same time. This accounts for the particularly poor worst-case performance of a single insertion for the Buffer Tree for the *LessSkewed* dataset.

A reasonable solution for the Buffer Tree is to use an anticipatory write, emptying its buffers before they become full. Aside from causing a prolonged period of slowed insertion processing, this solution has a qualitative consequence on Buffer Tree performance. Specifically, the Buffer Tree would no longer be I/O-optimal, ostensibly slowing overall insertion processing performance. See the Buffer Tree paper for details [3].

Table 1 Insertion durations (in ms)

Data org.	Var (*Skewed*)	Max. (*Skewed*)	Var (*LessSkewed*)	Max. (*LessSkewed*)
T1SM	2608	22,704	1796	10,433
SMM	1556	18,153	1709	15,667
LSM	2923	40,539	1488	18,916
Buffer Tree	24,095	101,918	116,269	451,203
Y-Tree	1315	3276	1675	2851
No Clustering	99	9321	185	6284
B+-Tree	678	2709	1849	2223

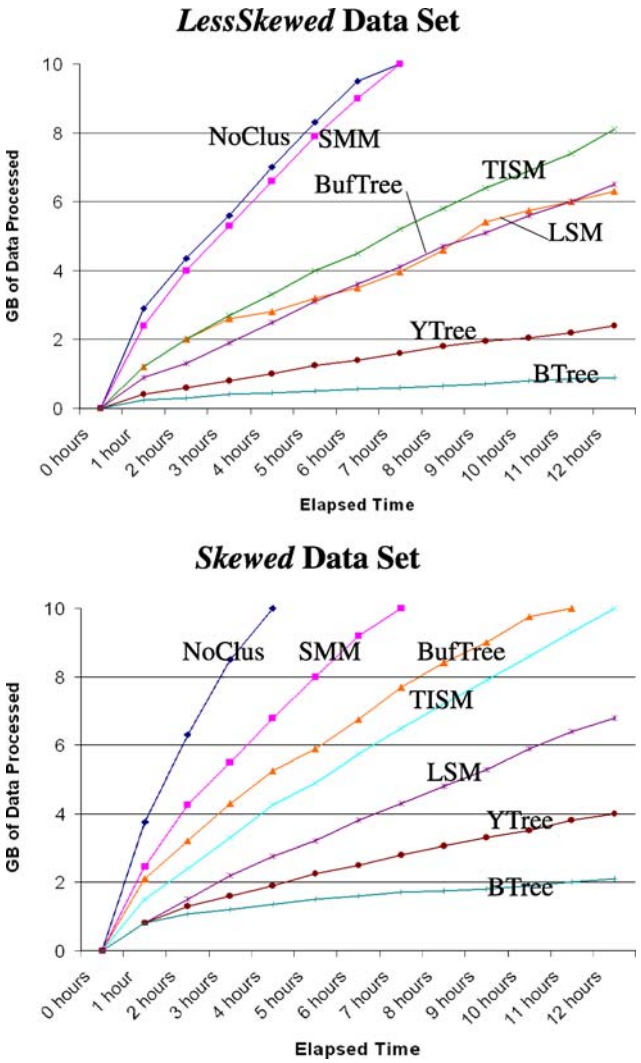

Fig. 11 Processing insertions for the *Skewed* and *LessSkewed* datasets

As mentioned above, poor worst-case insertion times make efficient concurrent query processing difficult, if not impossible. The Buffer Tree, in particular, can have such an extensive series of cascading buffer flushes that it is far from obvious how one could even guarantee that a record is not missed (or read multiple times) during concurrent query evaluation. Also, the reorganization represents a significant time window during which additional insertions cannot be processed.

5.6 Queries

5.6.1 Experiments

We test query performance for each organization by pausing the insertion process described above every 2 min to query the data. (Note that query time is not added to the construction time limit.) In this Section, we discuss raw query performance and consider concurrent queries and insertions later. We test three types of queries:

1. Q_1. This is an exact-match point query, searching for all records with a given key value. The key value is randomly chosen from those already inserted.
2. Q_2. This is a "medium" sized range query. The expected result set size is 1/50,000 of the database based on the range.
3. Q_3. This is our largest range query. The expected result set size is 1/1000 of the database based on the range.

In Figs. 12 and 13, we plot the time required to evaluate the different query types for each of the data organizations. For the un-clustered organization (NoClus), we run only Q_1 queries. Under this organization, it is not possible to efficiently evaluate range queries because each tuple in the result set would require a separate seek. We plotted 30-point (or 1 h) running medians to smooth the graph and remove most of the anomalies from the data. This smoothing explains why the graphs do not start at 0 seconds when the structure is small enough to be buffered entirely in memory.

Note that some of the graphs abruptly stop (particularly for the B+-Tree and the Y-Tree). This is a result of the limited size of data available for querying. After 12 h of insertions, the B+-Tree and the Y-tree contain only 1 and 2.5 GB, respectively, of the 10 GB insertion set. In these cases, applying insertions until the file reaches the 10 GB limit is impractical. We project that it would take more than 2 weeks to build a 10 GB B+-Tree using our experimental configuration.

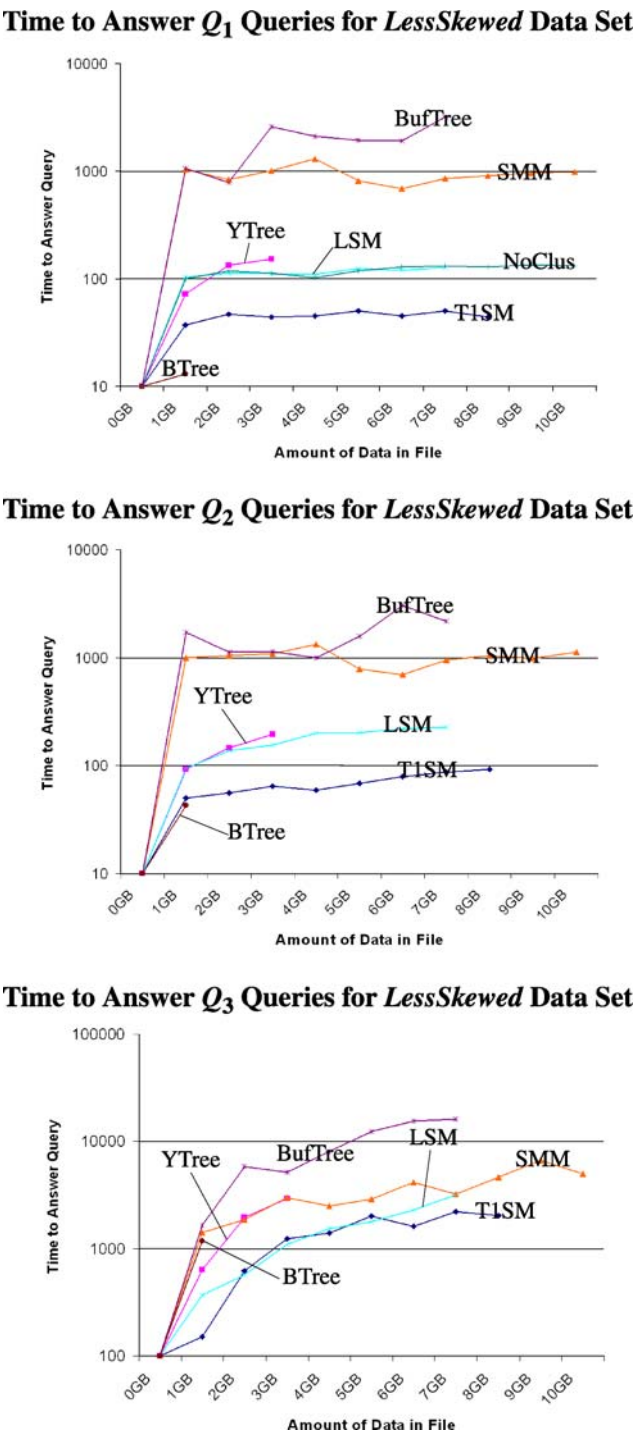

Fig. 12 Query processing speed, in milliseconds

5.6.2 Discussion

T1SM is by far the fastest organization tested for the small (Q_1) and medium-sized (Q_2) queries on the *Skewed* dataset, but it is outdone by the B+-Tree on the *LessSkewed* dataset. In general, however, we believe that T1SM would be slightly outperformed by a well-tuned B+-Tree over point queries for most datasets, if insertions are rare. The reason that T1SM is not outperformed by the B+-Tree on the *Skewed* dataset for our experiments is that while insertions are confined to a few hot spots, the queries are distributed uniformly and are issued only periodically. Thus, the LRU buffer associated with the B+-Tree adapts to the insertion pattern, and is generally ineffective for handling queries. If queries

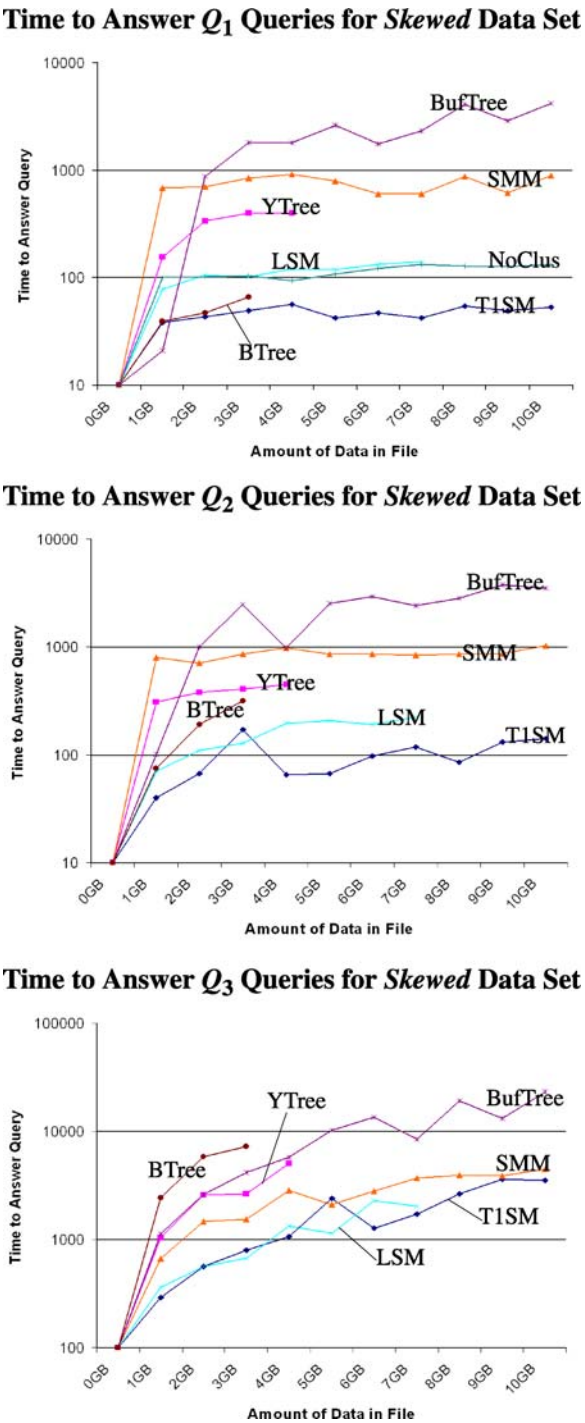

Fig. 13 Query processing speed, in milliseconds

were issued in isolation, this effect should disappear. In the case of the *LessSkewed* data, more of the B+-Tree is actively receiving insertions. As a result, more memory is allocated to buffering its upper levels than to its leaves. This is the ideal situation for evaluating the smaller, random queries of the experiment: the distribution of queries matches the distribution of insertions more closely for the *LessSkewed* data than for *Skewed* data. This match makes query processing more efficient. The Y-Tree shows a similar tendency toward better performance over the *LessSkewed* dataset.

Both the Buffer Tree and the SMM are not generally well-suited for point queries. The Buffer Tree is slow because of the substantial on-disk buffer (up to several MB). A query may require many such buffers to be read. The SMM is slow because a dozen or more individual trees might need to be searched to find the desired key value during point query evaluation. We note that the suggestion made by the designers of the SMM of including a bitmap or filter (that is, some sort of secondary index for each tree) in order to indicate which trees have which key values would be helpful for speeding point query evaluation. However, maintenance of a secondary index can be expected to slow update processing.

For larger queries, (especially queries of type Q_3) T1SM is always at least as fast as any of the other organizations. This performance is again attributable to the clustering provided by the partition abstraction. However, two points warrant further discussion. First, the LSM-Tree's performance is identical to that of T1SM for Q_3 queries. In fact, the performance of the LSM-Tree and T1SM for larger range queries would be very hard to improve upon, as for both we found that the median Q_3 query processing time was typically within 20% of what would be required to simply read the equivalent amount of data sequentially from disk. For smaller queries, the LSM-Tree is slow because searching multiple components hurts performance significantly. With larger queries, however, this (fixed) time is amortized by the time required to perform long, sequential scan.

Second, the significant performance improvement of the SMM for Q_3 queries is noteworthy. Its performance eventually approaches that of TISM for large range queries as the database grows. As with the LSM-Tree, a fixed SMM cost becomes insignificant for larger queries. While the SMM must search many trees in order to evaluate a query, the time required is fixed, and amortized by scan time of large queries.

5.7 Mixed workloads

Perhaps just as significant as the insertion performance described in Sect. 5.5 and the query performance described in Sect. 5.6 is the ability of a data organization to handle a insertions and queries simultaneously. In this section, we first measure the heaviest query load each data organization can handle given a minimum insertion rate. We then explore TISM's performance in more detail, and compare its performance with that of the B+-Tree on a mixed workload over a wide range of query and insertion rates. At the end of this section, we consider the PE file's performance on spatial data, demonstrating its ability to conform to other data types.

5.7.1 Experiments: Maximum query load with a given insertion rate

In our experimental setup, we preload 4 GB of data from the *LessSkewed* dataset into each of the seven organizations (the

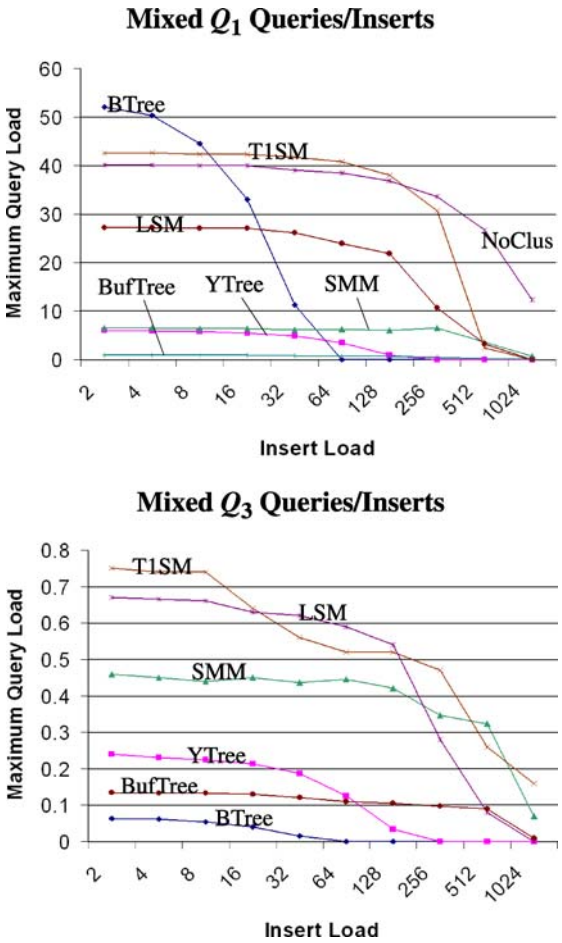

Fig. 14 Maximum insert/query loads per second for type Q_1 (point queries) and Q_3 (range queries)

B+-Tree required nearly 5 days to do this). We then insert data at various rates: 2, 4, 8, . . . , 1024 inserts per second. For each insertion rate, we interleave as many queries as possible, subject to the constraint that the insertion rate is maintained. We perform two 20-min experiments using this setup: one using Q_1 queries, and the other using Q_3 queries. Because of the inability of the NoClus method to handle large range queries, we do not consider it in the second experiment.

The rates (in queries per second) that each organization is able to handle are presented in Fig. 14. These results show how queries and insertions interact to affect the performance of each of the different organizations. Again, the graphs represent the maximum load each organization can handle.

5.7.2 Discussion

Interestingly, all organizations, except the Buffer Tree and the Y-Tree, take turns having the best performance depending on the experimental parameters. Not surprisingly, for point queries (Fig. 14a) with a light insertion load, the B+-Tree works best. However, its performance degrades quickly with increasing insertion rate. When the insertion rate is more than eight insertions per second, TISM is superior to the B+-Tree. At more than 80 insertions per second, the B+-Tree has absolutely no additional capacity with which it can process insertions, making it unusable. The non-clustered organization with a secondary Y-Tree (NoClus) is a close second to TISM for the medium to heavy insertion loads, and actually outperforms TISM for the heaviest insertion loads. The Y-Tree, the Buffer Tree, and the SMM all show relatively poor Q_1 query processing performance regardless of insertion load.

For large range queries, Q_3, TISM is again able to consistently handle the heaviest loads (Fig. 14b). The LSM-Tree has close (and occasionally better) performance until around 500 insertions per second, which is not surprising considering its raw query performance matched TISM for large range queries (see Sect. 5.6). However, as the insertion rate increases, the inferior insertion performance of the LSM-Tree hurts overall performance. The opposite is true of the SMM organization. It has inferior query performance, but better insertion processing performance compared to T1SM. Thus, it becomes very competitive with TISM for heavier insertion loads.

5.7.3 Experiments: Effects of insertions on query latencies for T1SM

We also consider the exact performance penalty that a high rate of insertions has on TISM's concurrent query response time. One concern is insertions occasionally trigger merges, splits or reorganizations of a PE file. These operations might not significantly affect the average query performance, but they might significantly hurt worst-case query performance, especially considering the effect of TISM's concurrency control mechanism (described in Appendix B). In this Section, we address a performance measure that was only partially treated by the insertion performance experiments described in Sect. 5.5.3.

We used the following setup to test the effect of concurrent insertions on TISM's query performance. A second Pentium PC functions as a client, transmitting a stream of inserts and queries to the original PC, which acts as a T1SM server, via Ethernet. T1SM is pre-loaded with 80 million 16-byte records. In this case, T1SM contains relatively small records, as it would if it were used as a secondary index (we run experiments with larger records in the next Section, using T2SM). The data are produced by 20 randomly positioned one-dimensional clusters, whose centroids slowly "wander" throughout the range 0 to 2^{31}. Each cluster produces integer keys distributed around the centroid using N, a normally distributed random variable with standard deviation 1. The value of each key produced by a cluster is (*cluster centroid*) $+ N \times 10000$. Thus, at any given time, there are 20 "hot spots" for data insertion. Insertion records for the latency experiments are generated in this way as well by the client PC.

Queries over the data are issued concurrently with the insertions, and are randomly generated in the following

manner. N, a normally distributed random variable with standard deviation of 0.0001 and a mean of 0, is used to generate the expected fraction of the database to be returned by the query, subject to the constraint that N is positive and not greater than 1. Average queries generated in this way return around 6000 tuples (or 96 KB of data). There is great variation in the number of tuples returned by the queries; many queries are effectively point queries and returned little or no data.

We test many combinations of concurrent query and insertions rates. To test a query rate Q with an insert rate I, we run an experiment where the client PC sends a stream of inserts and queries to the server PC for 10 min. The insert rate and query rate are each held constant for a second. After 1 s, they fluctuate, such that the query and insert rates for the next second are governed by random variables having a Poisson distribution, with means Q and I, respectively. In this way the rates averaged Q and I over the 10 min, but within those 10 min they are allowed to fluctuate substantially, as one would perhaps expect in reality under a heavy workload.

For each of these 10-min experiments, we measure the average elapsed time between the time the server receives the query, and the time the server totally determines the result set. The result is not transmitted to the client, so that network costs are not a factor in the results. If queries and inserts back up and the server is not able to recover substantially during the 10 min of the experiment, the server is said to have failed the experiment.

For a comparison, we perform the same experiments on our B+-Tree implementation. For simplicity, we do not implement a concurrency control mechanism in the B+-Tree. This eliminates the possibility of implementation bias, and yields "idealized" B+-Tree performance. Reads and writes are allowed to occur simultaneously without locking. To avoid creating an inconsistent B+-Tree, leaf pages are never modified due to insertions. To simulate update costs, however, leaf pages that should be updated are read into memory, and then written out unchanged. Thus, the tree is never updated and conflicts never occur, but most of the costs associated with updates are measured.

The results of these experiments are shown in Fig. 15. This Figure depicts the mean query latency for each of the 10 min experiments. The results clearly indicate that while T1SM occasionally encounters long-lived insertions (as described in Sect. 5.5), its query latencies, even in an insert-heavy environment, are very small overall. Moreover, T1SM significantly outperforms the B+-Tree in each of the tests. The reason that the cells in the two plots of Fig. 15 do not darken monotonically as one moves up and to the right is that each cell corresponds to a single, 10-min experiment, and the distribution of queries and inserts is very bursty. Thus, some variation is expected, since during one experiment, the B+-Tree or T1SM may simply be very unlucky, and encounter a really bad burst that it is unable to recover from. Still, we believe that despite this variation, the plots do clearly highlight the different characteristics of the two schemes.

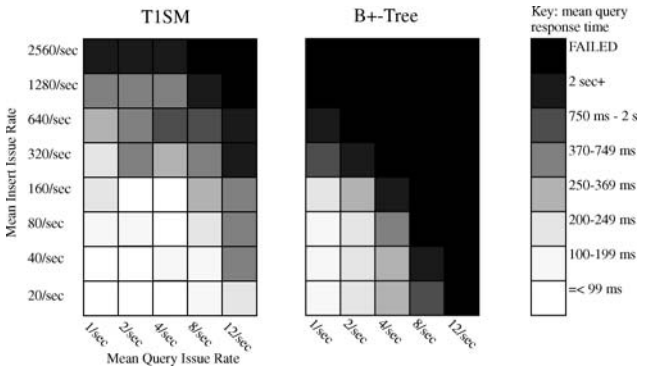

Fig. 15 Comparison of results for concurrent query/insert experiments between T1SM and the B+-Tree

5.7.4 Experiments: Effects of insertions on query latencies on spatial data using T2SM

We also conduct tests similar to those described in the previous section on spatial data. In this case, we compare the performance of the Taboose 2 Storage Manager (T2SM [18]), a PE file customized for spatial data, with that of an R-Tree. To perform this comparison, we consider the following two datasets.

- The *Cluster* dataset. This synthetic dataset is generated as follows. One-hundred two-dimensional Gaussian clusters are scattered randomly across a 20 by 20 field. The standard deviation of each cluster along each dimension is 1. Individual data points are allotted to clusters in a skewed manner, according to a Poisson distribution, with a mean point count of 300,000. As a result, there are a total of 30 million points in the dataset. Each tuple in the dataset contains a search key and a pointer consuming a total of 36 B. Thus, the structures are effectively being tested as secondary indices, as in the previous section. The total amount of data for this dataset is 1.08 GB. A visualization of a small, random sample of this dataset is shown in Fig. 16, left.
- The *Random Walk* dataset. This is another synthetically generated dataset. One hundred particles are scattered throughout an area 10,000 units wide. Each particle has a certain amount of energy, distributed according to a Poisson distribution, with a mean of 5. The particles are tracked for 100,000 clock ticks, where at each clock tick the particle moves in a random direction. The distance travelled per tick is governed by a Poisson distribution, with a mean distance equivalent to the particle's energy. There are thus 10 million total tuples. Each tuple has 120B of information about a particle's position and motion at a given instant. Tuples are organized by position. The total size of the dataset is 1.2 GB. It is visualized shown in Fig. 16, right. Clearly, the spatial distribution of data is very skewed.

As in Sect. 5.7.3, the goal of these experiments is to test the effect of heavy insertion loads on concurrent query processing. To do this, we use a very similar setup to the one

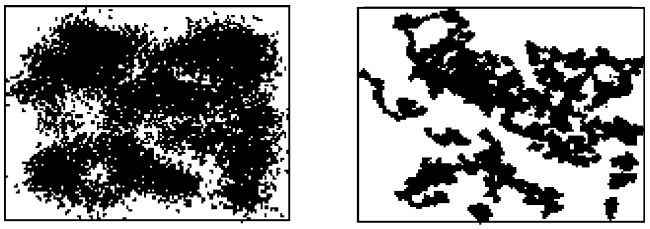

Fig. 16 A visualization of a subset of the Cluster (*left*) and Random Walk (*right*)

used in Sect. 5.7.3; one PC is used as a T2SM server, and another PC generates a stream of insertions and queries to be processed by the server.

For both datasets, queries are issued concurrently with various insertion loads, just as in the last Section. Queries are randomly produced, and uniformly scattered over the dataspace. The size of the region queried in a given query is produced by a normally distributed random variable, N, with standard deviation of 0.0001. This absolute value of this variable determines expected fraction of the database to be returned by the query. For queries generated in this way, the average result set is 2404 tuples (or 87 KB of data) for the Cluster dataset, and 870 tuples (or 104 KB of data) for the Random Walk dataset. The aspect ratio of the queries is generated as follows. A normally distributed random variable N with standard deviation 1 is used to produce a number, n. If n is greater than 0, then the aspect ratio of the query region is $1 + n$. If n is less than 0, then the aspect ratio is set to $1/(1 - n)$. Thus, the "average" query is a square, with more rectangular queries becoming rarer as the degree of "rectangularness" increased.

The R-Tree is fully packed using the STR algorithm [21] to produce a high-quality organization that has minimal problems with overlap and strangely shaped regions. As in Sect. 5.7.3, we sidestep the problem of concurrency control in the R-Tree by not allowing insertions to actually change the R-Tree organization; all they do is to cause disk head movements. Thus, this R-Tree also demonstrate "idealized" concurrency control performance. It is worth mentioning that when using standard, incremental update techniques like forced re-inserts, modern variations on the R-Tree structure can require more than 5 disk I/Os per insertion [11]. Thus, our test is simulation a very idealized case for the R-Tree.

The results of our experiments are summarized in Fig. 17 for the Cluster and the Random Walk datasets. Again, the absolute latencies experienced using T2SM are acceptable, and significantly better compared to those of the R-Tree.

5.8 Summary of experimental results

It would be unreasonable to expect that any one of the data organizations that we studied would be preferred for use in every situation. The fact that 30 years of research has gone into the development of database access methods (and that the problem is still far from solved) testifies to the breadth

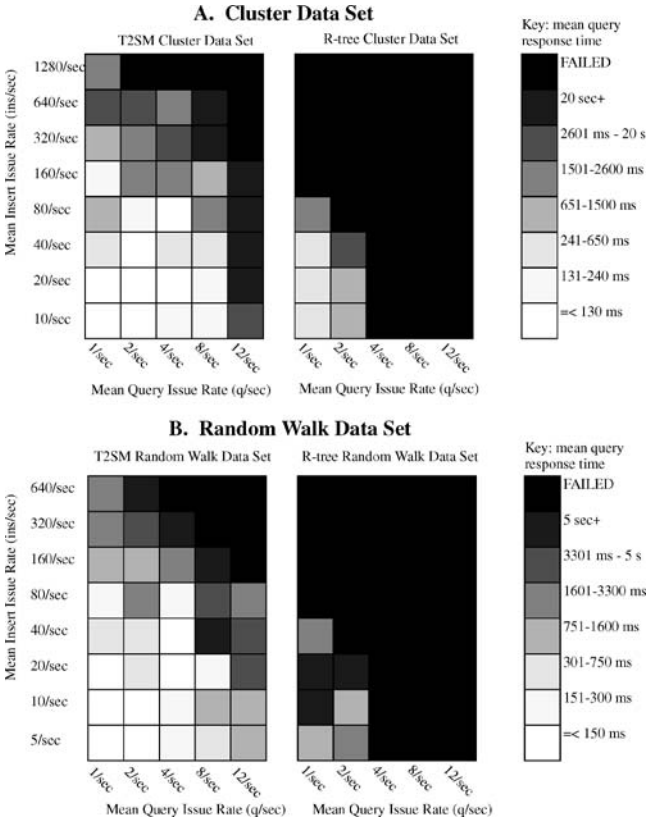

Fig. 17 Comparison of results for concurrent query/insert experiments for T2SM and an R-tree on the Cluster **A** and Random walk **B** Datasets

and variety of the different requirements that can be placed on a storage organization for even one-dimensional data, and to the difficulty of finding a single, ultimate solution to the problem.

With this in mind, we attempt to summarize in Table 2 the relative performance that we observed for each of the data organizations tested during our benchmarking. For each of several categories, we rank the various data organizations from one to seven, and we also give a final, average ranking for each. The rankings are necessarily somewhat arbitrary, and do not tell the whole story. For example, a B+-Tree would deserve to be ranked first for concurrent point queries processed with a light insertion load, but would deserve to be ranked last for point queries processed concurrently with a heavy insertion load; we gave it a ranking of three overall in this category. The ranking can also mask significant gaps in relative performance. For example, there is a large gap in medium range query performance between T1SM and the LSM-Tree on one hand and the SMM on the other; however, the former occupy the top two slots and SMM is ranked third. Finally, many other issues would be important to fully explore, but are beyond the scope of the paper in general and the rankings in particular. For example, what is the effect of record size on the different organizations? The effect might be significant, as fewer (or more) records fit on a page. We considered variable record sizes in Sects. 5.7.3 and 5.7.4, but not in a systematic fashion. Other factors we did

Table 2 Summary of results

	T1SM	SMM	LSM	NoClus	BTree	YTree	BufTree
Raw insertion speed	3	2	5	1	7	6	3
Insertion latency	5	4	5	1	2	3	7
Point query evaluation	2	6	3	3	1	5	7
Range query evaluation	1	3	1	7	6	4	5
Point queries + inserts	1	5	3	1	3	6	7
Range queries + inserts	1	3	1	7	6	4	4
Avg ranking	2.16	3.83	2.83	3.33	4.16	4.66	5.5

not systematically consider include maintaining secondary indexes, and using different organizations *as* a secondary index.

Thus, we include the rankings with some reluctance. But despite these issues, we believe that Table 2 provides the most succinct summary of the different strengths and weaknesses of the organizations we tested. Based on our experiments, T1SM showed good to excellent performance in each of the categories considered.

6 Related work

Indexing has been studied for decades in the database community, and hence there are literally thousands of papers related to the work described in this paper. At the highest level, most indexing and data organization schemes rely either on *randomization* (as in hashing) or on *linearization*, where "similar" data are mapped to the same page on disk (as in the B+-Tree).

Broadly speaking, methods relying on randomization are dominated by those based on various hashing methods. Widely cited examples of hashing are *linear hashing* [22] and *extendible hashing* [10], and the various external hashing schemes described by Knuth [20]. We have not considered hashing in this paper, because the goal of this work is the development of methods for drastically reducing disk seeks in an insert or update heavy environment with concurrent, analytic queries. Hashing is usually not useful for reducing random seeks. Randomization generally destroys locality, and hence range queries over a hashed file can require more than one seek for each record returned (this is debilitating for large queries). To overcome this problem, hashing could be performed using the most significant bits in the search key (for example, using extendible hashing), but such a data organization would still require one seek per data insertion. Hashing is clearly very useful in many situations, but probably not in the environments that we consider.

On the other hand, methods relying on linearization map similar key values into the same disk block in order to reduce random I/Os during query processing, and are therefore more relevant to our work. Classic examples of such organizations are the B+-Tree [8] and its close cousin, the R-Tree [14]. The PE file also falls into this category. These data organizations are typically more useful for answering analytic or range queries, because similar records are written to the same page. This organization of records into pages reduces seeks during query evaluation, because only one seek is required to retrieve all of the records on a page, which may all be needed to answer a given range query.

Although all of the data organizations falling into this category are somewhat related to the PE file, a primary goal of the PE file is keeping pages clustered intelligently on disk, so that all of the pages needed to answer a query can be read in a single, sequential scan. This difficult issue has been studied before in the context of linear data organizations. Examples include the Lomet's *bounded disorder file* [23], and O'Neil's *SB-Tree* [28]. The bounded disorder file organization partitions the data space into large ranges of key values that can be stored contiguously on disk, with each range is organized as a hashed file. The SB-Tree can be seen as a precursor to the LSM-Tree [29] (also partially developed by O'Neil), in that the SB-Tree maintains multi-page runs of blocks for sequential operations over the data. For spatial data, a related idea was explored by Seeger et al. [34]. More recently, Tao and Papadias [35] have considered a B+-Tree variant that allows multi-page blocks, where the size of the blocks are chosen based on previous query characteristics. These multi-page blocks can be stored contiguously on disk, though Tao and Papadias did not consider the lower-level issues that arise when varying-sized runs of pages must be managed on disk (such as fragmentation). Broadly speaking, also related are methods for performing monolithic reorganizations of data on disk. If blocks are not kept clustered in a dynamic manner, they can be reorganized during a dedicated reorganization period [31, 37].

None of the above methods address the problem that each database update can still require one or more random seeks. Avoiding such random seeks during update processing was a primary goal in the design of the PE file. Several other structures have been proposed that can be used to reduce the cost per update to a fraction of a seek in a linear data organization. Many of these structures have been described and benchmarked in Sect. 5. Those specifically considered in Sect. 5 are the Y-Tree [17], the LSM Tree [29], the Buffer Tree [3], and the Stepped Merge Method [16]. Some variations on the ideas benchmarked in Sect. 5 for use with different types of data exist in the literature. Van den Bercken et al. [6] and Arge et al. [4] both describe variations on the Buffer Tree for spatial data. Muth et al. [25] describe a variation on the LSM-Tree for use with temporal data. In the introduction of the paper, we mentioned some

early variations on the idea of an exponential/logarithmic file that allow very fast updates/insertions [30]. This idea has also been extended to data structures for other data, such as multidimensional data [2].

Finally, we mention the work from the operating systems community on log-structured file systems [26, 33]. In this work, the idea is to speed writes and reduce seeks by treating the file system as a log, and always appending writes or updates to the files in the file system to the end of the log. Log-structured file systems are not closely related to the issues discussed in this paper, since they are meant to speed updates in a general-purpose file system, not in a DBMS where information about data semantics and query patterns must also be considered. However, the idea of organizing data as a log in a DBMS in order to speed the processing of update-heavy workloads has been considered previously, and forms the basis for the LSM-Tree, for example.

7 Conclusions and future work

In this paper, we presented the PE file organization. The PE file is a generic template for data organization that is suitable for use with many different types of data (for example, data organized on a single numerical attribute, or spatial or temporal data). The PE file is designed specifically for use in an environment that is insertion or update intensive, but where queries (especially over large ranges of key values) must be processed concurrently. This situation is common in modern data recording environments where updates must be installed and available immediately.

A key design principle of the PE file is that it uses disk seeks only sparingly, since their cost is increasing exponentially over time, with respect to storage capacity and scan rate. The goal of minimizing seeks, especially random seeks, naturally results in the linearization of on-disk data structures, which is done with exponential file data organizations [5, 30]. The problem with such data organizations is that, unlike traditional, page-based organizations (like the B+-Tree), the cost of query processing is a function of database size. We solve this problem by partitioning the dataset, thereby bounding query processing cost.

We experimentally demonstrated that the performance of the PE file performing point queries is very competitive with that of page-based data organizations, which are expressly designed for such workloads. We also experimentally demonstrated that the PE file performing larger range queries is very competitive with exponential data organizations, which are optimized for such workloads. Overall, the PE File is competitive over a wide range of workloads when compared to data organizations optimized for those workloads. The continued evolution of hard disk technology should make such performance advantages clearer in the future.

The most obvious avenue for future work is an extension of the PE file for a multi-disk environment, as one would naturally expect to find in the type of large-scale storage application that the PE file is designed for. The most significant question that must be addressed when extending the PE file to a multi-disk environment is how to handle the natural seek versus sequential transfer trade-offs that become even more difficult to address across multiple disks. For example, for large updates and data objects where the sequential transfer time dominates, it makes sense to stripe the PE file partitions across many disks, to make use of parallel I/O. However, for smaller data objects and updates, the PE file partitions should be kept on only one or perhaps a few disks, in order to conserve seeks (because each disk holding part of a data object must use a seek to perform an update). In addition, a PE file could be striped horizontally or vertically across disks, or the PE file could make use of both options simultaneously. Because of the careful way in which the PE file manages seeks and sequential I/O, the PE file offers an exciting opportunity to develop efficient and adaptive schemes for managing data in a multi-disk environment. This is an exciting avenue for future research.

Appendix A: Analytically expected insertion speed

We assume a uniform pattern of insertions in our analysis. (Uniform insertion is often the most difficult case for an indexing scheme to deal with because buffering becomes less effective.) We wish to build a database S_{DB} bytes in size, with an available insertion-processing buffer memory of $S_{DB}/1{,}000$, and a partition size of S_{part}.

In a PE file under these conditions, the number of disk seeks performed is two for every $S_{\text{part}}/1000$ bytes of inserted data (because this is the size of L_1 for each partition, assuming buffer memory is evenly distributed among them). Whenever the L_1 buffer pool is empty, we must merge/pack L_1 from some partition with one or more of that partition's upper levels. Each merge/pack requires two seeks: one to read, and one to write the appropriate levels. Note that this is independent of record size.

Now, let us compare this against a standard, hierarchical structure, like a B+-Tree or R-Tree. For such a structure, we could buffer leaf nodes in main memory, but this would be essentially useless because the probability that a single insertion falls into a buffered leaf page is very small (1/1000, under our assumption of uniform insertion). Instead, it would make sense to pin in memory as many of the upper levels in the tree as possible. Assuming all internal levels are pinned, there would be two seeks *for each insertion:* one to read the leaf and one to write it.

What about the total amount of data read/written to build a PE file? As we indicated above, each time some partition's L_1 is full, a merge/pack for that partition is required. This means that there are $(S_{\text{DB}} \times 1000)/S_{\text{part}}$ merge/pack operations performed to build the database, each of which has a cost of:

$$2 \times \sum_{l=1}^{M} (F^l/2) \times (1/F^{(l-1)}) \times (S_{\text{part}}/1000)$$
$$= 2 \times \sum_{l=1}^{M} (F/2) \times (S_{\text{part}}/1000) = \frac{M \times F \times S_{\text{part}}}{1000}$$

The intuition here is that on average, a partition level is 1/2 full. Each level has an exponentially decreasing probability of being merged into (because only one out of F merges into L_i cause a merge into L_{i+1}) but an exponentially increasing cost for being merged into (due to exponentially increasing size). These two exponents cancel one another out. The two is included in the expression because we must both read *and* write to merge/pack. To complete the above expression, we must ask: given the number of disk-based partition levels M, what is the multiplicative factor F? This is easily computed using the fact that $F^M =$

	General Formulas		Expected Values for Example			
	Total I/O (Bytes)	Total seeks	#seeks	Total I/O (Bytes)	seek time (hours)	seq. I/O time (hours)
PE file	$M \times F \times S_{DB}$	$\frac{2000 \times S_{DB}}{S_{size}}$	4.0e6	3.96e12	11.1	55
Hierarchical File	$\frac{S_{DB} \times S_{page} \times 2}{S_{rec}}$	$\frac{S_{DB} \times 2}{S_{rec}}$	9.38e8	7.68e12	2606	107

Fig. 18 Insertion costs for PE and hierarchical files

1000. This is because the size of a partition is 1000 times the size of L_1 For example, if M is 3, then L_2 is ∼31 times as large as L_1, and L_3 is ∼31 times as large as L_2, or 1000 times as large as L_1. Solving for F and substituting into the above expression gives us the read/write cost per merge. Because there are $S_{\text{DB}} \times 1000/S_{\text{part}}$ merge/pack operations performed in total, the total number of bytes read/written is then approximately $M \times F \times S_{\text{DB}}$ bytes read and written, in total.

We also consider the number of bytes read/written for a hierarchical structure per insert. Again, ignoring page splits and assuming the upper levels of the tree are pinned, we must read/write a leaf node for each insert. With a record size S_{rec} and a page size S_{page}, the total number of bytes read/written is $(S_{\text{DB}}/S_{\text{rec}}) \times S_{\text{page}} \times 2$.

In the final analysis, seek costs in a hierarchical file are possibly debilitating when processing a large number of insertions. We evaluate the above expressions for a 60 GB DB to be stored on a single disk, for both the hierarchical structure and the PE file, assuming 60 MB of memory for buffers, a partition size of 30 MB (with $M = 3$) for the PE file, and a 8 KB page size for the hierarchical file. We assume sequential read and write rates of 20 MB/s and a seek time of 10 ms. Records are 128B each, for a total of 383 million records. The results of the evaluation are shown above in Fig. 18. The most striking result is the huge time requirement for the seeks in the hierarchical file. Assuming two seeks per insertion (which is conservative, especially for spatial indices; one modern method requires on the order of five per insert to build a small database [11]) this works out to more than 3 months of time required to perform disk arm movements. Note that in both cases, time required for sequential I/O is negligible when compared to a hierarchical file's seek cost.

Appendix B: PE file concurrency control and recovery (CC&R)

In this Appendix, we briefly outline CC&R for the PE file.

Appendix B1: Concurrency control

The PE file relies heavily on a shadow paging method for CC&R. Whenever a partition needs to be written in its entirety (that is, when there has been a flush into the highest level of the structure, or during the type of reorganization described in Sect. 3.4), the new version of the partition is always written to a new, empty location on disk. Because of this, queries can be processed concurrently on the old version, while the new version is being constructed and written.

For simplicity, we only discuss single-insertion transactions. Multiple insertion transactions will likely require more complicated techniques, or higher level locking of the entire structure.

At the highest level, when a new insert is to be processed, the insertion thread first obtains a write lock on the global, in memory structures depicted in Fig. 4. Then, there are three cases:

- Most frequently, the insertion is simply written to an available main memory buffer page (which is done entirely in memory) and the lock is released almost instantaneously.
- Less frequently, a merge from a lower partition level to a higher level is required to free up the partition's main memory buffer pages. In this case, the partition which is to be merged and packed returns its main memory buffer pages to the L_1 buffer pool, and the global structures of Fig. 3 are unlocked. Then, a merge/pack with the contents of the partition and its main memory buffer pages are performed. The partition to be operated on is locked to exclude reads and writes over it during the merge/pack.
- In the relatively rare case that one or more partitions must be written in their entirety to process the insertion, a boolean flag called *ReorgFlag* is set to true, and the partitions which are to be reorganized are locked to block subsequent insertions into them. The flag *ReorgFlag* signifies that a reorganization is in progress. Subsequent insertions into other, non-locked partitions may proceed, but they are not allowed to trigger another reorganization while the current one is in progress. Then, each partition to be merged/packed and re-written is processed in turn. Because of the shadow paging method which is used, queries may still read the shadow copy of a partition, even as a new version of it is under construction. After the partition has been merged/packed and written to its new position, the global structures are updated to indicate this change, and the shadow copy of the partition is freed.

Appendix B2: Recovery

Recovery is facilitated through the use of logical logging. To facilitate fast logging, batches of transactions should be committed simultaneously in a group commit (which would allow only a fraction of an I/O to be used per insertion).

Obviously, checkpointing must be performed periodically to avoid arbitrary growth in the size of the log. To perform a checkpoint, we copy the contents of all but the highest level of all partitions to a separate location on disk. This checkpoint is not too costly, because the total size of the lower levels of partition is only a small fraction of its overall size.

In the case of a failure, we have two cases for each partition: either we were physically writing that partition at the time of the failure, or we were not. We now describe the actions required for recovery in each case.

- If we were *not* writing a given partition at the time of the failure, then we need to reconstruct that partition's L_1 (its main memory level), by replaying all updates to that partition from the log, beginning with the last time that a flush from L_1 to L_2 happened, for that partition.
- If we were writing that partition at the time of the failure, there are two subcases. If we were writing the partition in its entirety, then we simply go back to the shadow copy of the partition, and then reconstruct L_1 for that partition, using the recent entries from the log. If instead we were performing a merge into one of the lower levels of the partition, then we have the most costly situation. Here, we must redo all of the updates to the partition, beginning with the last time it was written out in its entirety. If it was not written out in its entirety since the last checkpoint, then we take its lower levels from the last checkpoint as a starting point, and redo all of the updates to it from the log.

Note that, using this recovery scheme, we never need to replay changes to a partition that occurred before the last time the partition was written in its entirety. This has the effect of truncating the length of the log naturally, and might be used to speed checkpointing.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM **31**, 1116–1127 (1988)
2. Agarwal, P.K., Arge, L., Procopiuc, O., Vitter, J.S.: A framework for index bulk loading and dynamization. In: Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001) pp. 115–127. Crete, Greece (2001)

3. Arge, L.: The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In: Algorithms and Data Structures, 4th International Workshop (WADS 1995) pp. 334–345. Kingston, Ontario, Canada (1995)
4. Arge, L., Hinrichs, K., Vahrenhold, J., Vitter, J.S.: Efficient bulk operations on dynamic R-trees. Algorithm Engineering and Experimentation, International Workshop (ALE-NEX 1999), Baltimore, MD, USA. January 15–16, pp. 328–348 (1999)
5. Bentley, J.L.: Decomposable searching problems. Information Processing Letters **8**(5), 244–251 (1979)
6. Bercken, Jochen Van den, Seeger, B., Widmayer, P.: A generic approach to bulk loading multidimensional index structures. In: Proceedings of 23rd International Conference on Very Large Data Bases (VLDB 1997) pp. 406–415. Athens, Greece, (1997)
7. Chen, P.M., Lee, E.L., Gibson, G.A., Katz, R.H., Patterson, D.A.: RAID: High-performance, reliable secondary storage. ACM Computing Surveys **26**(2), 145–185 (1994)
8. Comer, D.: The ubiquitous B-Tree. ACM Computing Surveys **11**(2), 121–137 (1979)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms, MIT Press, Massachusetts (1992)
10. Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: Extendible hashing—A fast access method for dynamic files. ACM Transactions on Database Systems **4**(3), 315–344 (1979)
11. Garcia, Y.J., Lopez, M.A., Leutenegger, S.T.: On optimal node splitting for R-trees. In: Proceedings of the 24th International Conference on Very Large Data Bases (VLDB 1998) pp. 334–344. New York City, New York, USA (1998)
12. Gray, J., Shenoy, P.J.: Rules of thumb in data engineering. In: Proceedings of the 16th International Conference on Data Engineering (ICDE 2000) pp. 3–12. vol. **3** San Diego, CA, USA.
13. Growchowski, E.G.: Emerging Trends in Data Storage on Magnetic Hard Disk Drives. Datatech (1998)
14. Guttman, A.: R-Trees: A dynamic index structure for spatial searching. In: Proceedings of 1984 SIGMOD Conference (SIGMOD 1984) pp. 47–57. Boston, Massachusetts (1984)
15. Hellerstein, J.M., Naughton, J.F., Pfeffer, A.: Generalized search trees for database systems. In: Proceedings of 21th International Conference on Very Large Data Bases (VLDB'95) pp. 562–573, Zurich, Switzerland (1995)
16. Jagadish, H.V., Narayan, P.P.S., Seshadri, S., Sudarshan, S., Kanneganti, R.: Incremental organization for data recording and warehousing. In: Proceedings of 23rd International Conference on Very Large Data Bases (VLDB 1997) pp. 16–25. Athens, Greece (1997)
17. Jermaine, C., Datta, A., Omiecinski, E.: A novel index supporting high volume data warehouse insertion. In: Proceedings of 25th International Conference on Very Large Data Bases (VLDB 1999) pp. 235–246. Edinburgh, Scotland, UK (1999)
18. Jermaine, C., Omiecinski, E., Yee, W.G.: Maintaining a Large Spatial Index with T2SM. In: Proceedings of the Ninth ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS 2001). Atlanta, GA, USA. (2001)
19. Kamel, I., Faloutss, C.: Hilbert R-tree: An improved R-tree using fractals. In: Proceedings of the 20th International Conference on Very Large Data Bases pp. 500–509. Santiago de Chile, Chile (VLDB 1994) (1994)
20. Knuth, D.E.: The art of computer programming, Vol. III: Sorting and Searching. Addison-Wesley, Reading, MA (1973)
21. Leutenegger, S.T., Edgington, J.M., Lopez, M.A.: STR: A simple and efficient algorithm for R-tree packing. In: Proceedings of the Thirteenth International Conference on Data Engineering (ICDE 1997) pp. 497–506. Birmingham, UK (1997)
22. Litwin, W., Hashing, L.: A new tool for file and table addressing. In: Proceedings of the Sixth International Conference on Very Large Data Bases pp. 212–223. Montreal, Quebec, Canada (VLDB 1980) (1980)
23. Lomet, D.B.: a simple bounded disorder file organization with good performance. ACM Transactions on Database Systems **13**(4), 525–551 (1988)
24. Lomet, D., Salzberg, B.: Access methods for multiversion data. In: Proceedings of the 1989 ACM SIGMOD Conference on the Management of Data (SIGMOD 1989) pp. 315–323. Portland, Oregon, (1989)
25. Muth, P., O'Neil, P.E., Pick, A., Weikum, G.: Design, implementation, and performance of the lham log-structured history data access method. In: Proceedings of 24rd International Conference on Very Large Data Bases (VLDB 1998) pp. 452–463. New York City, New York, USA (1998)
26. Neefe, J.M., Roselli, D.S., Costello, A.M., Wang, R.Y., Anderson, T.E.: Improving the performance of log-structured file systems with adaptive methods. In: Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP 1997) pp. 238–251. St Malo, France (1997)
27. O'Neil, J.E., O'Neil, P.E., Weikum, G.: The LRU-K page replacement algorithm for database disk buffering. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD 1993) pp. 297–306. Washington, DC (1993)
28. O'Neil, P.E.: The SB-tree: An index-sequential structure for high-performance sequential access. Acta Informatica **29**(3), 241–265 (1992)
29. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica **33**(4), 351–385 (1996)
30. Overmars, M.H.: The design of dynamic data structures. Springer-Verlag, LNCS p. 156 (1983)
31. Park, J.S., Sridhar, V.: Probabilistic model and optimal reorganization of B+-Tree with physical clustering. IEEE Transactions on Knowledge and Data Engineering **9**(5), 826–832 (1997)
32. Pollari-Malmi, K., Soisalon-Soininen, E., Ylönen, T.: Concurrency control in B-Trees with batch updates. IEEE Transactions on Knowledge and Data Engineering **8**(6), 975–984 (1996)
33. Rosenblum, M., Ousterhout, J.K.: The design and Implementation of a log-structured file system. ACM Transactions on Computer Systems **10**(1), 26–52 (1992)
34. Seeger, B., Kriegel, H.-P.: The Buddy-Tree: an efficient and robust access method for spatial data base systems. In: Proceedings of the 16th International Conference on Very Large Data Bases (VLDB 1990) pp. 590–601. Brisbane, Queensland, Australia (1990)
35. Tao, Y., Papadias, D.: Adaptive Index Structures. In: Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002) pp. 418–429. Hong Kong, China (2002)
36. Tyson, A., The LSST Collaboration: Large synoptic survey telescope: Overview. In: Proceeedings of SPIE; International Society of Optical Engineering 4836, pp. 10–20 (2002)
37. Zou, C., Salzberg, B.: On-line reorganization of sparsely-populated B+trees. In: Proceedings of the 1996 ACM S1GMOD International Conference on Management of Data (SIGMOD 1996) pp. 115–124. Montreal, Quebec, Canada (1996)