Cost Models for Overlapping and Multiversion Structures

YUFEI TAO
Hong Kong University of Science and Technology
DIMITRIS PAPADIAS
Hong Kong University of Science and Technology
and
JUN ZHANG
Hong Kong University of Science and Technology

Overlapping and *multiversion* techniques are two popular frameworks that transform an ephemeral index into a multiple logical-tree structure in order to support versioning databases. Although both frameworks have produced numerous efficient indexing methods, their performance analysis is rather limited; as a result there is no clear understanding about the behavior of the alternative structures and the choice of the best one, given the data and query characteristics. Furthermore, query optimization based on these methods is currently impossible. These are serious problems due to the incorporation of overlapping and multiversion techniques in several traditional (e.g., financial) and emerging (e.g., spatiotemporal) applications. In this article, we reduce performance analysis of overlapping and multiversion structures to that of the corresponding ephemeral structures, thus simplifying the problem significantly. This reduction leads to accurate cost models that predict the sizes of the trees, the node/page accesses, and selectivity of queries. Furthermore, the models offer significant insight into the behavior of the structures and provide guidelines about the selection of the most appropriate method in practice. Extensive experimentation proves that the proposed models yield errors below 5 and 15% for uniform and nonuniform data, respectively.

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

General Terms: Theory

Additional Key Words and Phrases: Database, temporal, spatiotemporal, index, overlapping and multiversion structures

1. INTRODUCTION

Supporting objects whose attributes change with time (i.e., *versioning* objects) is crucial for a large number of applications. As an example, consider a banking

This work was supported by grants HKUST 6081/01E and 6197/02E from Hong Kong RGC.
Authors' addresses: Y. Tao, Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong; email: taoyf@cs.ust.hk; D. Papadias, Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong; email: dimitris@cs.ust.hk; J. Zhang, Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong; email: zhangjun@cs.ust.hk.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2002 ACM 0362-5915/02/0900-0299 $5.00

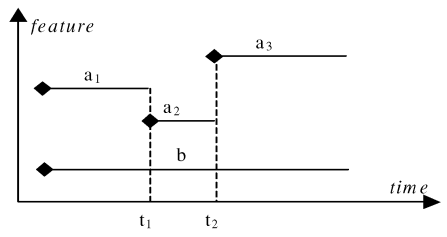


Fig. 1. Representation of versioning objects.

system that records the historical changes of account balances occurring as a result of withdrawals, deposits, or money transfers. Old versions of the records are not removed since possible queries may occur regarding any time in history, and the DBMS should provide efficient access paths to all recorded versions. Such *versioning databases* constitute the core of many temporal, spatiotemporal, decision-making, and online analytical systems.

We use the term *features* to refer to the time-varying attributes of versioning objects, which are best modeled as intervals in the feature–time space. Figure 1 shows an example for a banking system. The vertical axis refers to account balances (i.e., the features), and the horizontal axis corresponds to transaction time. Intervals a_1, a_2, and a_3 represent the balance changes of account a: one withdrawal at timestamp t_1 and one deposit at timestamp t_2. No change occurs to account b during the demonstrated period. Notice that we represent records as semiclosed intervals to emphasize that the valid period of a record does not include the last timestamp, when a new record becomes valid. In the sequel, we say that a record is *alive* during its valid period, and *dead* outside it. For example, record a_2 is alive in interval $[t_1, t_2)$.

An important type of processing in versioning databases involves *range-interval queries* (*interval queries*, for short), which consist of two predicates: the time interval of interest and a *feature range* in the *feature universe*. For the previous example, the feature universe is the range defined by the minimum and maximum possible balances. As another scenario consider traffic control systems that monitor and record movements of vehicles. In this case, the feature universe is a 2-D region that involves all the locations to which any vehicle can ever travel. The records retrieved by a query must be alive during the time interval and have their features in the feature range. Instances of range-interval queries are: "*Find the accounts with balances greater than \$1000 during March 2001*," and "*Which cars appeared in the campus yesterday 8 am to 6 pm?*" When the time predicate involves only a single timestamp, the query is called a *range-timestamp query* (*timestamp query*, for short).

Access methods for versioning databases have been extensively studied in temporal and spatiotemporal databases. Most existing methods are based on *multiple logical-tree structures* (MLTS). A MLTS maintains several logical trees, each of which is an *ephemeral structure* suitable for indexing objects at a single timestamp. To avoid excessive space storage, consecutive logical trees may share common branches so that these branches are stored only once. The *overlapping* and *multiversion* techniques are two popular frameworks for converting ephemeral structures (such as B-trees, R-trees [Guttman 1984], linear quadtrees [Gargantini 1982], etc.) into efficient temporal and spatiotemporal access methods.

Despite the large number of MLTS that have been proposed, little work has been carried out on their performance analysis. Existing models mainly focus on the overlapping and multiversion B-trees, merely discussing their asymptotic optimality with respect to timestamp queries. This is, however, insufficient for practical use for several reasons. First, in most cases asymptotic performance does not accurately reflect the actual cost. Second, interval queries, which are more frequent in practice, are not discussed. Third, existing analysis is only applicable to structures based on B-trees and cannot be used for other MLTS.

In this article, we provide an analytical framework that can be employed for any MLTS provided there exists a cost model for the corresponding ephemeral structure. For instance, in order to analyze the performance of MLTS based on R-trees, we only need to incorporate the corresponding R-tree models into our framework. The proposed models can accurately predict (1) tree sizes, (2) timestamp and interval query performance, and (3) query selectivity. The formulae are based only on the properties of the raw data and the underlying file system, and hence do not require knowledge about the structures of the trees. Furthermore, they are applicable for any data distribution and variable agility, and can be used in the presence of LRU buffers. Our analysis can be employed to tune the structural parameters in order to optimize the performance. Moreover, it provides significant insight into the behavior of alternative structures, and leads to important guidelines about the selection of the most appropriate method given the data and query characteristics.

We deal with partially persistent trees [Salzberg and Tsotras 1999] (i.e., updates can be applied at the current timestamp only), not including structures (e.g., Lanka and Mays [1991]) where updates are allowed at any timestamp in history. Furthermore, to facilitate analysis we make the following assumptions: every object has the same probability to issue a change at each timestamp; and the number of insertions approximates the number of deletions at each timestamp. Note that these conditions are satisfied in a wide range of applications (e.g., systems dealing with bank accounts, university transcripts, employee records, vehicle movements, multimedia objects, etc.). The rest of the article is organized as follows. Section 2 surveys overlapping and multiversion structures and describes in detail the two frameworks using B-trees as the ephemeral structures. Section 3 presents the cost models for methods based on B-trees, and Section 4 discusses their extensions to support general structures in real-life scenarios (LRU buffers and arbitrary data distributions). Section 5 presents an extensive experimental evaluation, and Section 6 concludes the article with future directions.

2. OVERLAPPING AND MULTIVERSION ACCESS METHODS

The overlapping technique was introduced in Burton and Huntbach [1985] and Carey et al. [1986] to produce a time and space efficient approach to file sharing. The idea was applied to B-trees in Burton et al. [1990] and R-trees in Xu et al. [1990] and Nascimento and Silva [1998]. The resulting structures were called overlapping B-trees (OVB-trees) and historical R-trees (HR-trees), respectively. Tzouramanis et al. [1999] extended OVB-trees by integrating pointers among leaf pages, which improved the so-called key-history queries in temporal databases. The technique was also applied to linear quadtrees [Tzouramanis et al. 2000a] and spatiotemporal data warehousing [Papadias et al. 2002]. In a survey paper Salzberg and Tsotras [1999] compared asymptotic performance of overlapping structures with other temporal access methods in terms of timestamp query performance, update costs, and structure sizes. Interval query performance was not discussed.

The earliest multiversion structure appeared in Easton [1986], who proposed the write-once B-tree (WOB-tree) for write-once-read-many (WORM) disks. Focusing on a combination of WORM (for historical data) and write-many-read-many (WMRM, for current data) disks, Lomet and Salzberg [1989] presented the time-split B-tree (TSB-tree), which, as analyzed in Lomet and Salzberg [1990], introduced less redundancy than WOB-trees, and thus reduced the index size considerably. Becker et al. [1996] optimized the multiversion framework (in terms of asymptotical performance for space and timestamp query cost) in their multiversion B-tree (MVB-tree), which is similar to the TSB-tree, but employs an important mechanism called the *version condition* (to be elaborated upon shortly). Varman and Verma [1997] discussed a variation of MVB-trees that reduced the size requirements by some constant factor.

Multiversion structures based on R-trees include BTR-trees to index bitemporal databases [Kumar et al. 1998], PPR-trees [Kollios et al. 2001], and MV3R-trees [Tao and Papadias 2001a] for spatiotemporal databases. Multiversion linear quadtrees were proposed for image processing in Tzouramanis et al. [2000b]. The concept was also applied in branched temporal databases [Jiang et al. 2000] and temporal aggregation [Zhang et al. 2001], respectively, to obtain BT-trees and multiversion SB-trees. Recently, Chien et al. [2002] adopted the technique for XML processing, and Tao et al. [2002] used it for aggregate processing of planar points. Despite the large number of structures, to the best of our knowledge there does not exist any work that estimates the sizes and performance of multiversion methods in terms of node accesses for interval queries.

In the rest of the section, we describe the overlapping and multiversion frameworks using B-trees as the ephemeral structure. Other MLTS can be constructed by applying the same transformation algorithms on the corresponding ephemeral structures.

2.1 Overlapping B-Trees

The idea behind OVB-trees is to maintain a separate B-tree for each timestamp in history, but allow consecutive trees to share branches as long as the

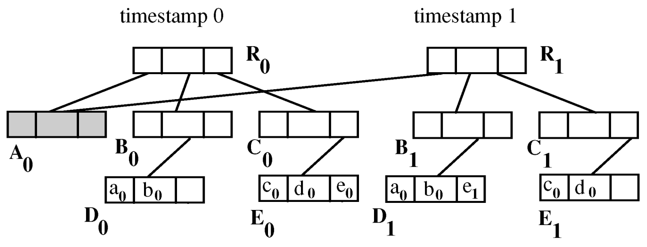


Fig. 2. An OVB-tree example.

underlying records do not change. Update algorithms are similar to those of B-trees, except that whenever a shared node is to be modified, we duplicate it to a new node where the changes are applied instead. Figure 2 illustrates part of an OVB-tree for timestamps 0 and 1. Assume that, at timestamp 1, account e changes from its previous balance e_0 to a new one e_1. Therefore, e_0 should be removed from the B-tree at timestamp 1, and e_1 should be inserted. In order not to affect the tree at timestamp 0, the removal of e_0 causes the duplication of E_0, creating node E_1. Similarly, the insertion of e_1 spawns new node D_1, which contains the entries of D_0 plus e_1. The changes propagate upwards, creating nodes B_1 and C_1. Notice that node A_0 is shared by both trees, indicating that none of the objects under A_0 issue any update at timestamp 1. Therefore, considerable space may be saved when the number of objects that change at each timestamp is relatively small. Note that such node duplication introduces data redundancy. For instance, separate records of objects a_0, b_0, c_0, d_0 are created even though these objects do not generate new versions at timestamp 1.

To keep track of the roots of the logical B+-trees, an OVB-tree maintains a *root table*, with one entry per root block. A timestamp query is directed to the corresponding B-tree and search is performed inside this tree only. Thus the query degenerates into an ordinary range query on B-trees and is handled very efficiently. An interval query involving several timestamps should search the corresponding trees of the related timestamps. Since a node can be pointed to by multiple parents, it is necessary to avoid duplicate visits to the same node via different parents, which can be achieved via "*positive and negative pointers*" described in Tao and Papadias [2001b]. For a timestamp query that returns K objects, OVB-trees achieve the optimal query cost of $O(\log(N_V/b) + K/b)$ node accesses, consuming, however, suboptimal space $O(N_V \log(N_V/b))$, where N_V is the total number of versions in history, and b the node capacity.

2.2 Multiversion B-Trees

In multiversion structures, each entry has the form $<key, t_{st}, t_{ed}, pointer>$ where t_{st} (the *insertion time*) denotes the time that the record was inserted in the databases, and t_{ed} (the *deletion time*) denotes the time that it was deleted.[1] For leaf entries, *key* denotes the feature of an object (e.g., the balances of accounts

[1]Such temporal information is unnecessary in overlapping structures, as each node contains entries of a single timestamp.

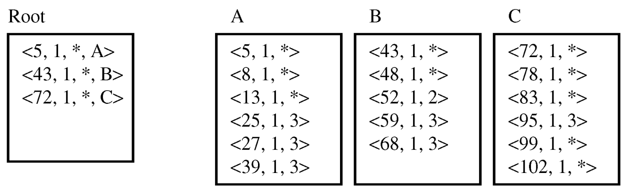


Fig. 3. An example of an MVB-tree.

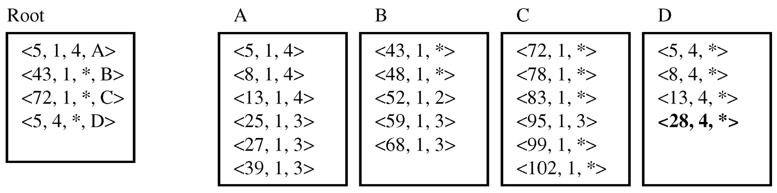


Fig. 4. Example of block overflow and version split.

or positions for vehicles). For an intermediate entry, *key* determines the minimum bounding range of features in the subtree alive in its lifespan $[t_{st}, t_{ed})$; its semantics follow that of the corresponding ephemeral structure. For MVB-trees [Becker et al. 1996], *key* equals the minimum value of features in the subtree (the bounding range can be derived by considering the value of *key* in the next entry).[2] The field *pointer* points to the actual record, or a node at the next level, for leaf and intermediate entries, respectively. When a new entry is inserted at timestamp t, t_{st} is set to t and t_{ed} to "*" (which denotes NOW). When an entry is logically deleted (due to an update), t_{ed} is changed (from *) to t. Entries with "*" as deletion time are referred to as *live entries*; otherwise they are *dead*. Figure 3 illustrates an example of an MVB-tree.

For each timestamp t and each node except the roots, it is required that none, or at least $b \cdot P_U$ entries are alive at t, where P_U is a tree parameter and b the node capacity (for the following examples $P_U = 1/3$ and $b = 6$). This *weak version condition* ensures that entries alive at the same timestamp are mostly grouped together in order to facilitate timestamp queries. A weak version underflow occurs if this condition is violated (e.g., due to deletion at the current time).

Insertions and deletions differ from those of the ephemeral structure (in this case, B-trees) in that overflows and underflows are handled differently. *Block overflow* occurs when an entry is inserted into a full node, in which case a *version split* is performed. To be specific, all the live entries of the node are copied to a new node, with their t_{st} modified to the current time. The value of t_{ed} of these entries in the original node is set to the deletion time as well (in practice this step can be avoided since the deletion time is implied by the entry in the parent node). In Figure 4, the insertion of <28,4,*> into node A at timestamp 4 (in the tree of Figure 3) will cause node A to overflow. A new node D is created to store the live entries of A, and A "dies" (notice that all * are replaced by 4) meaning that it will not be modified any more in the future. A new entry

[2]For MVR-trees (i.e., the multiversion structure based on the ephemeral structure R-tree), K is the minimum bounding rectangle (MBR) of the live entries.

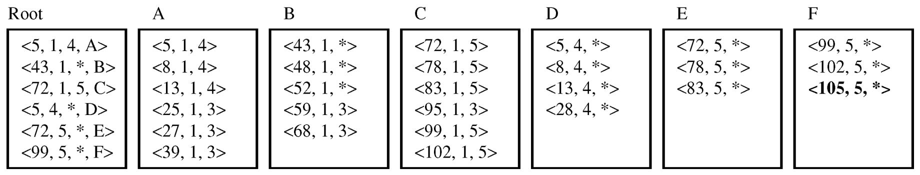


Fig. 5. Example of strong version overflow and key split.

<5,4,*,D> (pointing to the new node) is inserted into the root node. When the root generates a version split, the new node of the split becomes the root of another logical tree. Note that, as with node duplication in OVB-trees, version splits also introduce data redundancy. In Figure 4, for example, new records for leaf entries with keys 5, 8, and 13 are created in node D even though they do not generate new versions at timestamp 4.

In some cases, the new node may be almost full after a version split so that a small number of insertions would cause it to overflow again. On the other hand, if it contains too few entries, a small number of deletions would cause it to underflow. To avoid these problems, it is required that the number of entries in the new node must be in the range $[b \cdot P_{SVU}, b \cdot P_{SVO}]$ after a version split. P_{SVO} (SVO stands for *strong version overflow*) and P_{SVU} (*strong version underflow*) are tree parameters whose tuning is discussed later (for the following examples assume $P_{SVU} = 1/3$ and $P_{SVO} = 5/6$). A strong version overflow (underflow) occurs when the number of entries exceeds $b \cdot P_{SVO}$ (becomes lower than $b \cdot P_{SVU}$). A strong version overflow is handled by a *key split,* which is a version-independent split according to the features of the entries in the block, and is processed in the same way as the ephemeral structure. In Figure 5, for example, <105,5,*> is inserted into node C in the tree. Node C is version split, followed by a key split, and nodes E and F are generated (spawning two new entries in the root). Note that the strong version condition is only checked after a version split; that is, it is possible that the live entries of a node are above $b \cdot P_{SVO}$ after subsequent insertions.

Strong version underflow is similar to weak version underflow, the only difference being that the former happens after a version split, whereas the latter occurs when the weak version condition is violated after deletion. In both cases a merge is attempted with the copy of a sibling node using only its live entries. If the merged node strong version overflows, a key split is performed. Assume that at timestamp 4 we want to delete entry <48,1,*> from the tree in Figure 3. Node B weak version underflows since it contains only one live entry <43,1,*>. A sibling, node C, is chosen and its live entries are copied to a new node, C'. The insertion of <43,4,*> into C' leads to another key split and finally nodes D and E are created (Figure 6). In Varman and Verma [1997], the merging process was improved to reduce the tree size.

Each root has a *jurisdiction interval,* which is the minimum bounding lifespan of all the entries in the root (these jurisdiction intervals are mutually disjoint). The processing of a (timestamp or interval) query starts by retrieving the corresponding roots whose jurisdiction intervals intersect the queried interval. Then the search is guided by *key*, t_{st}, and t_{ed} down to the leaves. As with

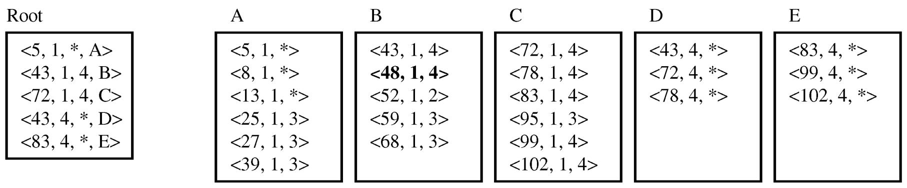


Fig. 6. Example of weak version underflow.

OVB-trees, a node may be visited more than once since it can have multiple parents at different timestamps. Techniques for avoiding multiple visits during the processing of interval queries in MVB-trees are discussed in Bercken and Seeger [1996].

As shown in Becker et al. [1996], for N_V versions produced by N objects, a multiversion structure consumes $O(N_V/b)$ space and answers timestamp queries in $O(\log(N_V/b) + K/b)$, where b is the node capacity and K is the number of records retrieved. Bercken and Seeger [1996] propose storing the so-called *backward pointers* in MVB-trees to improve interval query performance. The improvements are achieved by a specialized algorithm that avoids most nonleaf node accesses. Specifically, to answer an interval query the algorithm only needs to visit those nonleaf nodes that are alive at the first query timestamp; the leaf nodes containing results at other timestamps are reached by following backward pointers between leaf nodes. With some adaptations this technique also applies to OVB-trees to obtain similar interval query behavior [Tzouramanis et al. 1999]. The backward pointer technique, however, does not extend to general structures (such as MLTS based on R-trees).

3. PERFORMANCE ANALYSIS OF MULTIPLE LOGICAL-TREE STRUCTURES

Objects in versioning databases are updated with different frequencies. For example, in traffic control systems, the positions of vehicles usually change much faster than the balances of accounts in banking systems. To capture this, we define the concept of *agility* for temporal datasets as in Definition 3.1. Datasets with higher agilities incur more updates and involve more space requirements.

Definition 3.1. Let N be the number of objects, and u the number of objects that issue updates at timestamp i. Then we define the data *agility* a_i at timestamp i as

$$a_i = \frac{u}{N}.$$

Without loss of generality, we assume that features of objects are distributed in a unit universe $[0, 1]^d$, where d is the dimensionality (for B-trees, $d = 1$). An interval query q can be represented as $q(q_k, q_t)$ to indicate its feature range q_k and temporal interval q_t; q_k is defined in the feature universe: for MLTS based on B-trees, it is a one-dimensional range; and for MLTS based on R-trees, q_k is a multidimensional region. The interval range q_t corresponds to the set of timestamps queried.

The problem definition is as follows. At the first timestamp (timestamp 1), the features of N objects are distributed in the unit range $[0, 1)^d$ by a certain distribution $DIST_1$. At each of the subsequent timestamps i ($i = 2, 3, \ldots, T$), $a_i\%$ of the N objects issue feature changes, where T corresponds to the total number of timestamps recorded. The object distribution at timestamp i is denoted as $DIST_i$, and may vary for different timestamps. The updates are such that each object has the same probability to produce changes, and each update involves a (logical) deletion (invalidating the record denoting its previous version) and an insertion (for the new version). The dataset is indexed with an overlapping or multiversion structure, and the objective is to predict the sizes of the structures and their expected performance in terms of node (or page) accesses.

Existing analyses of MLTS [Becker et al. 1996; Varman and Verma 1997; Salzberg and Tsotras 1999] often assume that there is only one object update (i.e., a new version) per timestamp which, however, is too restrictive in practice. Consider, for example, the monthly salaries (objects in video/multimedia frames), where a timestamp corresponds to a month (frame). Obviously, multiple salaries (objects) may change at the same timestamp. One update per timestamp is simply a special case (i.e., where agility equals $1/N$) of our general problem definition and it is covered by our analysis. Furthermore, the agility of a dataset turns out to have a very significant impact on the behavior of MLTS. Our work constitutes the first systematic approach that relates dataset agility to MLTS performance.

The analysis proceeds as follows. In this section we present cost models for OVB- and MVB-trees assuming that (1) there is no buffer (i.e., focusing on node accesses), (2) $DIST_i$ is uniform for all timestamps, and (3) a_i does not vary throughout the history (i.e., $a_i = a$ for all $1 \leq i \leq T$). Although these assumptions may not be very realistic, they allow us to elaborate the essential characteristics of the alternative structures, and they serve as the basis for further discussion. Next, we remove these assumptions and discuss how to generalize the framework to other access methods, using the R-tree as an example. Table I lists the main symbols that are used frequently in our derivation. Some symbols have not appeared so far, but are elaborated shortly.

3.1 A Unified Cost Model

Similar to the representation of queries, we define a pair of ranges $s(s_k, s_t)$ for each node s in OVB- or MVB-trees. The *feature range* s_k corresponds to the minimum bounding range of all the entries in s, and the s_t is the period when s is valid in history. For a node in MVB-trees, s_t encloses the lifespans of all the entries in s. For OVB-trees, where the lifespans of the entries are not explicitly stored, s_t is defined as the period between the time that s is created and the time that it is duplicated. Since $DIST_i$ is uniform for all timestamps i, the structures of each logical tree in OVB- or MVB-trees remain approximately the same as the suitable clustering of the objects differs very little for each timestamp.

Nodes in MLTS are created in an "evolving" manner. That is, after the logical tree for the first timestamp is built, trees for subsequent timestamps are created by generating necessary nodes from the previous trees. The fact that an update

Table I. List of the Primary Symbols Used

Symbols	Definition
N	number of objects in the initial dataset ($\approx$ number of objects alive at each timestamp)
N_V	total number of versions
a_i	agility at timestamp i
$DIST_i$	object distribution at timestamp i
T	number of all the timestamps in history
b	capacity of a node
f	fanout of a node
h	height of a logical tree
$s(s_k, s_t)$	bounding feature and time ranges of a node s
$q(q_k, q_t)$	feature and time ranges of query q
K_i	total number of nodes at level i
E_i	evolution rate at level i
f_1	number of entries alive at one timestamp in an MVB-tree node
M_i	number of level i nodes alive at one timestamp in the MVB-tree
v_i	average number of level i version splits in the MVB-tree at one timestamp
V_i	total number of level i version splits in the MVB-tree in history

involves a deletion followed by an insertion and that every object has the same probability to issue changes leads to two important observations: live nodes at the same tree level receive approximately the same number of insertions (deletions) at each timestamp; and the number of live entries in a node remains roughly constant throughout its lifespan (because the number of insertions a node receives approximates that of deletions received at each timestamp).

As a result, after the first logical tree is constructed, node duplication and version split become the major types of structural changes for OVB- and MVB-trees, respectively. This is supported by our experiments: starting from the second timestamp, the number of key splits and merges (weak version underflows and strong version overflows/underflows) is significantly smaller (less than 5%) than the number of node duplications (version splits) for OVB- (MVB-) trees. Therefore, in the analysis, we may assume node duplication (version splits) to be the only type of structural changes for OVB- (MVB-) trees without introducing significant error. This allows us to focus on the factors that have the greatest impact on query performance.

If a node s_2 is created from a previous node s_1 through duplication (version split), then we say that s_1 *evolves* into s_2. To represent how fast the evolution proceeds, we define the *evolution rate* in Definition 3.2. As shown shortly, the evolution rate of nodes at a particular level of an OVB- or MVB-tree does not change significantly through history; hence, we use the notation E_i to denote the evolution rate for level i. Higher values for E_i indicate that new nodes are created with shorter cycles (smaller lifespans), resulting in larger trees.

Definition 3.2. Let M_i be the average number of live nodes of a particular tree level i at a timestamp. If on average, n_i new nodes of the same level are created at the next timestamp, then the evolution rate E_i for level i is

$$E_i = \frac{n_i}{M_i}.$$

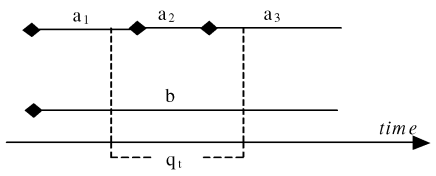


Fig. 7. Time evolution of nodes.

Notice that, in answering a query q, node $s(s_k, s_t)$ will be visited if and only if it intersects $q(q_k, q_t)$; that is, s_k intersects q_k and s_t intersects q_t. In other words, the probability that s will be visited is identical to the probability that $s(s_k, s_t)$ intersects $q(q_k, q_t)$, which we refer to as $prob(s, q)$. Let $prob_{feature}$ and $prob_{tm}$ denote the probability that the feature and temporal ranges intersect. Since the feature universe and the time dimension are independent, we have:

$$prob(s, q) = prob_{feature} \cdot prob_{tm}. \tag{3.1}$$

The estimation for $prob_{feature}$ has been studied in multidimensional access methods. If d is the dimensionality of the feature universe then:

$$prob_{feature} = \prod_{i=1}^{d} \left(s_k^i + q_k^i\right), \tag{3.2}$$

where s_k^i and q_k^i are the extents along the ith dimension for s_k and q_k, respectively [Theodoridis and Sellis 1996]. For the special case when the dimensionality of the feature universe is 1, the estimation for $prob_{feature}$ becomes:

$$prob_{feature} = s_k + q_k. \tag{3.3}$$

On the other hand, $prob_{tm}$ is closely related to the evolution rate of nodes. Let P_i be the total number of level i nodes whose lifespans intersect q_t and K_i be the total number of level i nodes ever created. We have:

$$prob_{tm} = \frac{P_i}{K_i}. \tag{3.4}$$

Assuming that the number of level i nodes alive at each timestamp is M_i, K_i can be estimated as

$$K_i = M_i + E_i M_i (T - 1), \tag{3.5}$$

where T denotes the total number of timestamps in history. The reasoning behind Equation (3.5) is that initially there exist M_i level i nodes, all of which are alive. Then, at each subsequent timestamp $E_i \cdot M_i$ nodes are created. When a new node is created, the previous node dies, so the lifespans of these two nodes are disjoint and continuous. Figure 7 shows a query whose temporal interval intersects the lifespans of four nodes, where a_2 and a_3 were created when a_1 and a_2 died, respectively.

The number of nodes whose lifespans intersect q_t is computed as follows. First, two nodes (a_1 and b) are alive at the first timestamp of q_t; then, during q_t, another two nodes (i.e., a_2 and a_3) are created; hence P_i equals 4. In general, there are $M_i = 2$ nodes alive at the first timestamp (time i) of q_t. Then, at each of the subsequent timestamps $i + j$ $(1 \leq j \leq |q_t| - 1)$, $E_i \cdot M_i$ nodes are created. Hence, we have the following estimation for P_i.

$$P_i = M_i + E_i M_i (q_t - 1). \tag{3.6}$$

Combining Equations (3.4) through (3.6), we have

$$prob_{tm} = \frac{1 + E_i(q_t - 1)}{1 + E_i(T - 1)}. \tag{3.7}$$

Extending Equations (3.1) and (3.3), we derive:

$$prob(s, q) = (s_k + q_k)\frac{1 + E_i(q_t - 1)}{1 + E_i(T - 1)}. \tag{3.8}$$

Recall that $prob(s, q)$ states the probability for a node to be accessed in answering query q; thus the expected number of node accesses $NA(q)$ is given by the equation:

$$NA(q) = \sum_{every\ node\ s} prob(s, q).$$

If $s_i(s_{ik}, s_{it})$ are the average range and temporal extents of nodes at level i, the above equation can be written as (3.9), where h denotes the height of a logical tree, K_i denotes the total number nodes at level i, and $prob(s_i, q)$ is given by Equation (3.8).

$$NA(q) = \sum_{i=0}^{h-1} [K_i \cdot prob(s_i, q)]. \tag{3.9}$$

In this work, we assume each node in the tree occupies a single disk page; hence Equation (3.9) also gives the expected number of disk accesses. Obviously, the equation can be easily adapted to general cases where a node can occupy multiple pages. This cost model, however, is "qualitative", in the sense that it must refer to the corresponding tree to obtain values for the relevant variables. In the sequel, we aim at representing s_{ik}, E_i, K_i, and h using the properties of the indexed dataset and the underlying file system.

3.2 A Cost Model for OVB-Trees

In this section, we present the derivation of the cost model for OVB-trees through several steps. In each step, we focus on rewriting a particular component in Equation (3.9) as the function of variables whose values are obtainable without referring to the actual tree.

Estimating h. The height of a B-tree that indexes N keys is estimated as in Equation (3.10), where f is the fanout of the tree. The commonly adopted value for f is $\ln 2 \cdot b_{split}$ [Yao 1978], where b_{split} is the number of entries a node contains when it splits.[3] For OVB-trees, $b_{split} = b$. Note that since the node

[3]As shown in Yao [1978], the fanout of a node in the B-tree does not depend on the concrete data distribution, but rather on the sequence of insertions. For example, the fanout is lowest if records are inserted in increasing order of their keys, and it is roughly the same for randomized insertions. Our cost models are based on randomized insertions because they are most common in practice.

capacity is decided by the page size of the underlying file system, the value of f is independent of the indexed dataset. Furthermore, since each logical tree indexes the same number of objects, the height of each tree is expected to be the same.

$$h = \lceil \log_f N/b \rceil + 1. \tag{3.10}$$

Estimating E_i. We start with the estimation for E_0, the evolution rate at the leaf level. Let us consider a leaf node s of a logical tree at an arbitrary timestamp i. Recall that s will be copied to a new node at timestamp $(i+1)$ if and only if any change (i.e., insertion or deletion) occurs in the node. For a dataset with agility a, the total number of changes per timestamp equals $2aN$ because each object update involves one deletion and one insertion. E_0 corresponds to the probability that a leaf node is affected by any of these $2aN$ changes. A leaf node contains on average N/f entries. Given an update, every node has the same probability f/N to be affected; thus, the probability for a node *not* to be affected by a single change is $(1 - f/N)$. Since all the changes are independent, the probability for a node not to be updated by any of these changes is $(1 - f/N)^{2aN}$. Thus we have:

$$E_0 = 1 - \left(1 - \frac{f}{N}\right)^{2aN}. \tag{3.11}$$

In general, the number of nodes at level i is N/f^{i+1}; hence the likelihood for a level i node to be affected by a change is f^{i+1}/N. Following the derivation of (3.11), we obtain:

$$E_i = 1 - \left(1 - \frac{f^{i+1}}{N}\right)^{2aN} \quad (0 \le i \le h-1). \tag{3.12}$$

If N is sufficiently large (which is true in practice), we have

$$E_i \approx 1 - (1-e)^{2af^{i+1}}.$$

As a side product of the estimation for E_i, we have the following lemma for the expected number of timestamps s_{it} that a node at level i remains alive in history.

LEMMA 1. $s_{it} = 1/E_i$.

PROOF. Consider a node s of level i that is created at timestamp k. Since the probability that s is copied at one timestamp is E_i, it follows that the probability that node s is valid for j timestamps (i.e., it is copied at timestamp $k+j$) is $(1-E_i)^{j-1} \cdot E_i$. Therefore, the expected number of timestamps that node s is valid in history is given by

$$\sum_{j=1}^{\infty} \left\{ \left[E_i (1-E_i)^{j-1} \right] \cdot j \right\}.$$

The above series converges to $1/E_i$. □

Estimating K_i. Since, at level i, the number of live nodes at one timestamp is N/f^{i+1}, the number of new nodes created at each timestamp is $E_i \cdot (N/f^{i+1})$.

Hence, the total number of level i nodes is

$$K_i = \frac{N}{f^{i+1}} + E_i \frac{N}{f^{i+1}}(T-1) \quad (0 \le i \le h-1). \tag{3.13}$$

Note that a corollary of Equation (3.13) is that we can estimate the size of an OVB-tree as

$$Size(OVB) = \sum_{i=0}^{h-1} K_i = \sum_{i=0}^{h-1} \left[\frac{N}{f^{i+1}} + E_i \frac{N}{f^{i+1}}(T-1) \right]. \tag{3.14}$$

Estimating s_{ik}. Now it remains to estimate s_{ik}, the average key range of nodes at level i. Notice that, since each logical tree is simply an ordinary B-tree, this estimation is directly obtainable from the analysis of B-trees. In fact, when *DIST* is uniform, the key ranges of nodes at the same level are roughly the same. Given that there are N/f^{i+1} nodes at level i in a B+-tree, we have

$$s_{ik} = \frac{f^{i+1}}{N} \quad (0 \le i \le h-1). \tag{3.15}$$

So far we have rewritten all the components of Equation (3.9) as functions of f, N, a, and T. The number of node accesses for OVB-trees is presented in Equation (3.16).

$$\begin{aligned} NA(q) &= \sum_{i=0}^{\lceil \log_f N/b \rceil} \left[\frac{N}{f^{i+1}} + E_i \frac{N}{f^{i+1}}(T-1) \right] \left(\frac{f^{i+1}}{N} + q_k \right) \frac{1 + E_i(q_t - 1)}{1 + E_i(T-1)} \\ &= \sum_{i=0}^{\lceil \log_f N/b \rceil} \frac{N}{f^{i+1}} \left(\frac{f^{i+1}}{N} + q_k \right) \left[1 + \left(1 - \left(1 - \frac{f^{i+1}}{N} \right)^{2aN} \right) (q_t - 1) \right]. \end{aligned} \tag{3.16}$$

Note that when a equals 0, the above equation degenerates into a cost model for conventional B-trees.

3.3 A Cost Model for MVB-Trees

In the sequel, we carry out a similar analysis for MVB-trees based on Equation (3.9).

Estimating h. Let f_1 be the average number of live entries at a single timestamp in node s. Note that f_1 is different from f, which equals the total number of entries in s. Thus, the height of a logical tree is given by Equation (3.17):

$$h = \lceil \log_{f_1} N/b \rceil. \tag{3.17}$$

Meanwhile, let M_i denote the average number of level i nodes that are alive at a single timestamp in a logical tree. The estimation for M_i is

$$M_i = \frac{N}{f_1^{i+1}} \quad (0 \le i \le h-1).$$

The estimation for f_1 deserves further elaboration. Recall that, in MVB-trees, if a node consists of only entries at the same timestamp, then the number of the entries cannot exceed $b \cdot P_{SVO}$; otherwise a strong version overflow occurs and the node will be key split. Hence, $f_1 = \ln 2 \cdot b_{split} = \ln 2 \cdot b \cdot P_{SVO}$.

Estimating E_i. We first present the estimation for E_0. A node s contains f_1 entries when it is created from a version split; thus s will receive $(b - f_1)$ insertions before it generates a version split, which in turn leads to the creation of a new node. At each timestamp, as there are $a \cdot N$ insertions, each leaf node can receive on average $a \cdot N/M_0$ insertions. As a result, s will generate a version split after $(b - f_1) \cdot M_0/(a \cdot N)$ timestamps. Since $M_0 = N/f_1$, the number of timestamps s_{0t} that a leaf level node remains alive before it is version split, can be estimated as

$$s_{0t} = \frac{(b - f_1)M_0}{aN} = \frac{b - f_1}{af_1}.$$

Let V_i be the total number (in history) of version splits at level i, and v_i the average number of version splits per timestamp at level i. For the leaf level, V_0 and v_0 are estimated as

$$V_0 = M_0 \frac{T-1}{s_{0t}} = \frac{aN(T-1)}{b - f_1}, \quad \text{and} \quad v_0 = \frac{V_0}{T-1} = \frac{aN}{b - f_1}.$$

Recall that the evolution rate is defined as the number of new nodes over the total number of live nodes at a timestamp. Since version splits are the only type of structural changes considered, we have

$$E_0 = \frac{v_0}{M_0} = \frac{af_1}{b - f_1}.$$

Whenever a leaf node generates a version split, an entry will be inserted into its parent node at level 1. Hence, at each timestamp, the average number of insertions at level 1 is v_0, and every level 1 node receives on average v_0/M_1 entries. Similar to our analysis above, a node at level 1 will generate a version split $(b - f_1) \cdot M_1/v_0$ timestamps after its creation. Therefore, the lifespan of the node, s_{1t} is

$$s_{1t} = \frac{(b - f_1)M_1}{v_0} = \frac{(b - f_1)^2}{af_1^2}.$$

The estimation for V_1 is

$$V_1 = M_1 \frac{T-1}{s_{1t}} = \frac{aN(T-1)}{(b - f_1)^2}.$$

In the same way, we obtain the following equations for nodes at higher levels.

$$s_{it} = \frac{(b - f_1)^{i+1}}{af_1^{i+1}}, \quad \text{and} \quad V_i = \frac{aN(T-1)}{(b - f_1)^{i+1}} \qquad (0 \le i \le h-1). \tag{3.18}$$

Hence, we have

$$v_i = \frac{V_i}{T-1} = \frac{aN}{(b - f_1)^{i+1}}, \quad \text{and} \quad E_i = \frac{v_i}{M_i} = \frac{af_1^{i+1}}{(b - f_1)^{i+1}} \quad (0 \le i \le h-1). \tag{3.19}$$

Finally, note that, as with Lemma 1 for OVB-trees, $s_{it} = 1/E_i$ also applies to nodes of MVB-trees at all levels.

Estimating K_i. Given that the total number of version splits at level i is provided by Equation (3.18), the total number K_i of nodes created through history is:

$$K_i = M_i + V_i = \frac{N}{f_1^{i+1}} + \frac{aN\,(T-1)}{(b-f_1)^{i+1}} \quad (0 \le i \le h-1). \tag{3.20}$$

As a corollary of Equation (3.20), the size of an MVB-tree can be estimated as

$$Size(MVB) = \sum_{i=0}^{h-1} K_i = \sum_{i=0}^{h-1}\left[\frac{N}{f_1^{i+1}} + \frac{aN(T-1)}{(b-f_1)^{i+1}}\right]. \tag{3.21}$$

Estimating s_{ik}. As mentioned above, a node contains f_1 live entries at the same timestamp. Therefore, replacing f in Equation (3.15) for OVB-trees with f_1, we obtain the following equation for the average key range of nodes at level i.

$$s_{ik} = \frac{f_1^{i+1}}{N} \quad (0 \le i \le h-1). \tag{3.22}$$

Equation (3.23) presents the final model, which predicts the node disk accesses for range-interval queries based on the properties of the indexed dataset and the underlying file system.

$$\begin{aligned} NA(q) &= \sum_{i=0}^{\lceil \log f_1 N/b\rceil}\left[\frac{N}{f_1^{i+1}} + E_i\frac{N}{f_1^{i+1}}(T-1)\right]\left(\frac{f_1^{i+1}}{N} + q_k\right)\frac{1+E_i(q_t-1)}{1+E_i(T-1)} \\ &= \sum_{i=0}^{\lceil \log f_1 N/b\rceil}\frac{N}{f_1^{i+1}}\left(\frac{f_1^{i+1}}{N} + q_k\right)\left[1 + \frac{af_1^{i+1}\cdot(q_t-1)}{(b-f_1)^{i+1}}\right]. \end{aligned} \tag{3.23}$$

Tuning P_{SVO}. Recall that a MVB-tree has several tree parameters, among which P_{SVO} (i.e., the strong version overflow threshold) has the most significant effect on the overall performance of the tree.[4] To understand this, observe from Equations (3.21) and (3.23) that both the tree size and query performance are very closely related to f_1, which, as mentioned earlier, can be approximated as $\ln 2 \cdot b \cdot P_{SVO}$. Our models provide useful insight towards choosing a good value for P_{SVO}. For example, according to Equation (3.21) setting a low P_{SVO} (which in turn makes f_1 smaller) usually reduces the tree size, by decreasing the term $(aN(T-1))/((b-f_1)^{i+1})$, which corresponds to the number of nodes created after the first timestamp, and will eventually dominate the other term N/f_1^{i+1} (i.e., the number of nodes that are alive only at the first timestamp). Intuitively, a smaller f_1 allows a node to receive more entries before it is version split, which reduces the total number of version splits, and hence data redundancy. However, if P_{SVO} is too low, the height of the tree (see Equation (3.17)

[4]Usually the other parameters are defined based on P_{SVO}, for example, $P_{SVU} = P_{SVO}/2$, $P_U = P_{SVO}/4$ [Varman and Verma 1997].

will become significantly higher, which, as shown in the experiments, may increase the total tree size if the number of nodes created at higher levels (during the entire history) exceeds the number of nodes diminished at lower levels. Furthermore, a high tree may compromise query performance. In fact, we observe that for any specific query parameters q_k and q_t, there exists an optimal P_{SVO} that minimizes the query cost $NA(q)$, and can be obtained by solving P_{SVO} from the equation,

$$\frac{d[NA(q)]}{df_1} \cdot \frac{df_1}{dP_{SVO}} = 0,$$

where $NA(q)$ is given by Equation (3.23). When $q_t = 1$ (timestamp queries), for example, the optimal P_{SVO} equals the maximum value 1, which means that strong version overflows can never happen. This is expected because a timestamp query only needs to search one logical B-tree; thus it is important to maximize the fanout f_1 of each logical tree. Queries with longer intervals have best performance with smaller P_{SVO} so that the resulting tree has less redundancy (due to the fact that version splits happen less often). These observations are verified by the experiments.

3.4 Selectivity Estimation

A record i with feature i_k and lifespan i_t, will be retrieved by a query q, if and only if $q(q_k, q_t)$ intersects $i(i_k, i_t)$. The probability $prob(i, q)$ for $i(i_k, i_t)$ and $q(q_k, q_t)$ to intersect is calculated according to Equation (3.8), except that the *evolution rate* of the objects now corresponds to the agility a of the dataset, and i_k is set to 0 because the feature of each entry indexed by an OVB- or MVB-tree contains only a single value. Hence, we have:

$$sel(q) = prob(i, q) = q_k \frac{1 + a(q_t - 1)}{1 + a(T - 1)}. \tag{3.24}$$

As a result, the number $NUM(q)$ of records retrieved by query q is estimated as

$$NUM(q) = prob(i, q) \cdot [N + aN(T - 1)] = N \cdot q_k \cdot [1 + a(q_t - 1)]. \tag{3.25}$$

3.5 Asymptotical Performance of Interval Queries

As mentioned earlier, traditional analysis on MLTS focused on their asymptotical performance for timestamp queries. Both structures achieve the optimal query cost, namely, $O(\log(N_V/b) + K/b)$ node accesses, and the space consumption is $O(N_V/b)$ (optimal) for MVB-trees and $O(N_V \log(N/b))$ (suboptimal) for OVB-trees, where N_V is the total number of versions produced by N objects throughout the history, K the number of versions retrieved, and b the node capacity. No result, however, exists for the asymptotical performance of either structure on interval queries. Particularly, since the MVB-tree is optimal (both in space and query cost) for timestamp queries, an interesting problem is to study whether it is also optimal for interval queries or, equivalently, whether it answers any interval query that returns K versions in $O(\log(N_V/b) + K/b)$ node accesses.

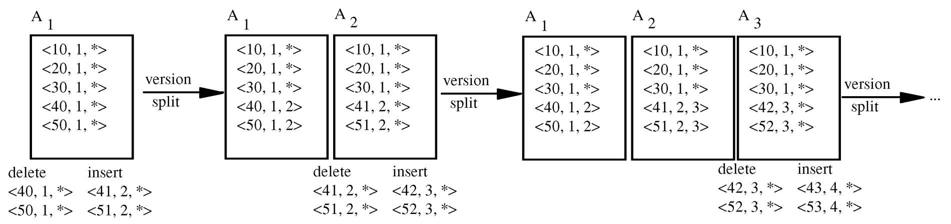


Fig. 8. The MVB-tree is not optimal for interval queries.

It turns out that the MVB-tree is not optimal in the worst case, because its interval query performance can be $\Omega(\log(N_V/b) + K/b + q_t)$, where q_t is the number of timestamps queried. To construct such a pessimistic case, we start with leaf node A_1 in Figure 8, which is created at timestamp 1 with 5 live entries (node capacity $= 6$). At the next timestamp, 2 deletions and 2 insertions are performed on A_1, which causes it to version split, creating node A_2 (still with 5 live entries). Notice that we do not modify the lifespans of the entries with keys 10, 20, 30 in both nodes, in order to emphasize that they are not new versions but simply redundant data resulting from the version split. Similarly, at the next timestamp, 2 deletions and 2 insertions are performed on node A_2, which version splits and spawns node A_3. This process is repeated at each timestamp i, so that a new node A_i is created and entries with keys 10, 20, 30 are never affected except for being duplicated in A_i.

Now we consider a query q with the key range [15, 25] and time interval [1, q_t], where q_t is no smaller than 1 and decides the length of the interval. Notice that in order to answer q, all nodes A_i $(1 \le i \le q_t)$ must be accessed because both their feature and temporal ranges intersect those of q, resulting in a processing cost of $\Omega(\log(N_V/b) + K/b + q_t)$ (the logarithmic factor comes from traversing the path from the root to leaves), which is worse than the optimal performance $O(\log(N_V/b) + K/b)$. The longer the query interval is, the more "suboptimal" the performance becomes.

Although this example is formulated with concrete values for tree parameters (specifically, fanout $f = 5$, node capacity $b = 6$, number of updates per timestamp $a \cdot N \ge 2$), it is easy to design similar cases for general settings. Specifically, starting from a leaf node A that has f entries, we may perform at least $(b - f + 1)$ deletions and $(b - f + 1)$ insertions on A (or nodes that evolve from A) at each timestamp, forcing it to version split and create a new node. Some entries in A are never affected by these updates (e.g., entries with keys 10, 20, 30 in Figure 8). Then, an interval query that retrieves only these unaffected entries has suboptimal query cost, because the number of versions retrieved is independent of the query interval length, whereas the number of node accesses increases linearly. Moreover, due to the similarity between nodes in OVB- and MVB-trees, such examples can also be constructed for OVB-trees, indicating that OVB-trees are not optimal for interval queries either.

After realizing the suboptimality of MVB-trees in the worst case, it is natural to investigate whether MVB-trees are still optimal for interval queries in the expected case. Note that the pessimistic example constructed earlier may not necessarily happen in the expected case, because it is rather unlikely that an object will remain unchanged for a long period of time if every object has the same probability to issue updates. To be specific, we may check if the estimated number of node accesses NA (produced from Equation (3.23) is indeed bounded by $O(\log(N_V/b) + K/b)$, where K denotes the number of versions retrieved (produced from (3.25). As shown shortly, when $N \cdot q_k \geq f_1$ the expected performance of MVB-trees is indeed optimal for interval queries by employing the query algorithm in Bercken and Seeger [1996], which, as mentioned in Section 2.2, only visits the nonleaf nodes whose lifespans intersect the first timestamp of the query, as well as all the leaf nodes whose feature and temporal extents intersect those of the query. The cost of visiting the nonleaf nodes corresponds to the logarithmic term $\log(N_V/b)$ of the optimal complexity. In the sequel, we show that the expected number NA_0 of leaf node accesses is bounded by $O(K/b)$.

To illustrate this, observe that, since $E_i = a \cdot f_1^i/(b - f_1)^i$, and $f_1 = \ln 2 \cdot b \cdot Psvo$, we have: $E_i = a \cdot (\ln 2 \cdot Psvo)^i/(1 - \ln 2 \cdot Psvo)^i$, leading to the fact that $E_i = O(a)$ (since P_{SVO} is a constant). This means that the evolution rate of MVB-trees at any level is as fast (in terms of complexity) as the data agility, which is somewhat against the intuition that nodes at higher levels should evolve more slowly than nodes at lower levels (i.e., with longer lifespans). In fact, notice that for some $Psvo$ (specifically when $\ln 2 \cdot Psvo > 1/2$) the evolution rates at higher levels can be even higher than those at lower levels. Formally, NA_0 is estimated from Equation (3.23),

$$\begin{aligned} NA_0 &= \frac{N}{f_1}\left(\frac{f_1}{N} + q_k\right)[1 + E_0(q_t - 1)] = \left(1 + \frac{N}{f_1}q_k\right)[1 + E_0(q_t - 1)] \\ &= 1 + \frac{N}{f_1}q_k + E_0(q_t - 1) + \frac{N}{f_1}q_k \cdot E_0(q_t - 1). \end{aligned} \tag{3.26}$$

Because $f_1 = O(b)$ and $E_i = O(a)$, NA_0 is bounded by (when $N \cdot q_k \geq f_1$)

$$NA_0 = O\left[\frac{N}{b}q_k + \frac{N}{b}q_k \cdot a(q_t - 1)\right].$$

Since $K = N \cdot q_k + N \cdot q_k \cdot a(q_t - 1)$, we have $NA_0 = O(K/b)$. Combining with the fact that the number of nonleaf node accesses establishes the logarithmic term $O(\log(N_V/b))$, the overall expected performance of MVB-trees is bounded by $O(\log(N_V/b) + K/b)$. Recall that this bound is achieved by assuming $N \cdot q_k \geq f_1$, which is required so that the term $E_0(q_t - 1)$ is absorbed by $(N/f_1) \cdot q_k \cdot E_0(q_t - 1)$ in the big-O complexity of Equation (3.26). Note that this extra condition ensures that the feature range q_k of the query is at least as long as the average feature range f_1/N of a leaf node. When this is not satisfied (consider, for instance, the pessimistic example given in Figure 8), the query cost may be bounded by $E_0(q_t - 1)$, which, intuitively, indicates that the number of versions retrieved by the query is not sufficient for justifying the nodes that evolved during the q_t and must be accessed by the query.

Next we conduct a similar analysis for the interval query performance of OVB-trees. However, instead of showing their expected optimality (which as can be conjectured is not true), we aim at quantifying the factor by which their expected performance deviates from the optimal cost. Similarly, we assume the algorithm proposed in Bercken and Seeger [1996] so that the total number of nonleaf node accesses corresponds to the logarithmic factor O(log(N_V/b)), allowing us to focus on the leaf node accesses. To facilitate analysis we assume N is large so that

$$E_0 = 1 - (1 - f/N)^{2aN} \approx 1 - e^{-2af}.$$

Furthermore, we observe that for typical values of f (on the order of 100 for regular disk pages), $e^{-2af} \approx 0$; thus $E_0 \approx 1$. From Equation (3.16) we have (similar to MVB-trees, assuming $N \cdot q_k \geq f$):

$$\begin{aligned} NA_0 &= \frac{N}{f}\left(\frac{f}{N} + q_k\right)[1 + E_0(q_t - 1)] = \left(1 + \frac{N}{f}q_k\right)[1 + E_0(q_t - 1)] \\ &= \mathrm{O}\left[1 + \frac{N}{f}q_k + (q_t - 1) + \frac{N}{f}q_k \cdot (q_t - 1)\right] = O\left[\frac{N}{b}q_k + \frac{N}{b}q_k \cdot \frac{1}{a}a(q_t - 1)\right] \\ &= \frac{1}{a}O\left[a\frac{N}{b}q_k + \frac{N}{b}q_k \cdot a(q_t - 1)\right] \\ &= \frac{1}{a}O\left[\frac{N}{b}q_k + \frac{N}{b}q_k \cdot a(q_t - 1)\right] = \frac{1}{a}O\left(\frac{K}{b}\right). \end{aligned}$$

Therefore, it is clear that the performance of OVB-trees is closely related to the data agility a (which is not the case for MVB-trees). As a increases, OVB-trees approach the optimal performance. Consider, for example, the extreme case where $a = 1$ (i.e., all objects change at every timestamp). Then, maintaining a separate B-tree at each timestamp is an optimal solution in the sense that the number of node accesses is linear to the number of object versions retrieved. In fact, as discussed in the next section and shown in the experiments, OVB-trees actually outperform MVB-trees beyond certain agility.

3.6 Predicting the Behavior of MLTS

The proposed models can answer two important questions: when it is worth using a MLTS instead of the *independent-tree implementation* (i.e., an individual B-tree for each timestamp), and which MLTS is preferable depending on structure size and query performance considerations. Regarding the first question, notice that when the agility exceeds a certain threshold (which we call the *degradation agility*), all the live nodes will be duplicated (version split) in an overlapping (multiversion) structure, at each timestamp; that is, both structures will degenerate into independent trees. To calculate the degradation agility, notice that a MLTS degrades completely when the evolution rate E_i approaches 1 for all levels, where E_i is defined in (3.12) and (3.19) for OVB- and MVB-trees, respectively. Solving these equations, we obtain the degradation agilities for OVB- and MVB-trees as follows.

$$\textit{deg-agility}(OVB) \approx \frac{-1}{N\log(1 - f/N)}. \tag{3.27}$$

$$\textit{deg-agility}(MVB) \approx \frac{b - f_1}{f_1} = \frac{1 - \ln 2 \cdot P_{svo}}{\ln 2 \cdot P_{svo}}. \tag{3.28}$$

For OVB-trees the estimated degradation agility is very low (less than 5% for our experimental settings), which severely limits their applicability. In order to intuitively explain this phenomenon, consider a situation where the average fanout of OVB-trees is $f = 100$. Even if one (out of 100) object in a node issues a change, the node needs to be copied (which leads to replication of all 100 entries). Furthermore, the update may lead to an insertion in another node, which will lead to duplication of that node as well. Therefore, in the worst case, even if less than 1% of the objects issue updates at a timestamp, an OVB-tree may degenerate to independent B-trees. Furthermore, the degradation agility becomes even lower for larger block sizes.

On the other hand, although the estimated degradation agility of MVB-trees is more than an order of magnitude higher (about 80% for our settings) this does not mean that MVB-trees are better than ephemeral B-trees up to this value of agility. Recall that each entry in a multiversion structure contains additional information about its lifespan, which lowers the node fanout. As a result, although an MVB-tree may have not degraded, above an agility threshold, which we call *size multitree point* (MTP, for short), it consumes more space than the independent-tree implementation. The size MTP can be predicted by solving the following equation, where the right part corresponds to the size of independent B-trees.

$$Size(MVB) = T \cdot \sum_{i=0}^{\lceil \log_f N \rceil - 1} \frac{N}{f^{i+1}},$$

Size(*MVB*) is given in Equation (3.21), and f is the fanout of a B- (or OVB-) tree. Solving this equation we obtain:

$$a_{MTP}(size) \approx \left(\frac{T}{f} - \frac{1}{f_1}\right) \cdot \frac{b_{MVB} - f_1}{T - 1} \approx \frac{b_{MVB}(1 - \ln 2 \cdot P_{SVO})}{f} \tag{3.29}$$

where b_{MVB} is the node capacity of the *MVB*-tree.

Similarly, for certain query parameters (q_k, q_t), we can also calculate the query MTP (the agility above which independent B-trees outperform the MVB-tree) by solving Equation (3.30). Although the query and size MTPs are not necessarily identical (the query MTP depends on the query parameters), their values, as shown in the experimental evaluation, are usually very close.

$$\begin{aligned} NA(q) &= q_t \cdot \sum_{i=0}^{\lceil \log_f N/b \rceil} \frac{N}{f^{i+1}} \left(\frac{f^{i+1}}{N} + q_k\right) \\ &\Rightarrow a_{MTP}(q_k, q_t) \approx \frac{\dfrac{1 + q_k \cdot N/f}{1 + q_k \cdot N/f_1} q_t - 1}{\dfrac{P_{SVO} \cdot \ln 2}{1 - P_{SVO} \cdot \ln 2}(q_t - 1)}. \end{aligned} \tag{3.30}$$

The above equation holds for $q_t > 1$ (i.e., interval queries). For timestamp queries, the query MTP is meaningless, because the independent-tree implementation (or OVB-tree) always outperforms MVB-trees due to their larger fanouts. For interval queries and large values of N, Equation (3.30) can be simplified to

$$a_{MTP}(q_k, q_t) \approx \frac{\frac{f_1}{f} q_t - 1}{\frac{P_{SVO} \cdot \ln 2}{1 - P_{SVO} \cdot \ln 2}(q_t - 1)}.$$

Furthermore, when q_t is sufficiently high (i.e., long interval queries), the query MTP converges to

$$a_{MTP}(q_k, q_t) \approx \frac{(1 - P_{SVO} \cdot \ln 2)\, f_1}{P_{SVO} \cdot \ln 2 \cdot f}.$$

For agilities below the query MTP, MVB-trees are usually more efficient for interval queries. Furthermore, the performance gain over OVB-trees can be obtained by considering Equations (3.16) and (3.23). Let NA_i(OVB) and NA_i(MVB) denote the number of node accesses at level i in answering query q with an OVB- and MVB-tree, respectively. Then, we have

$$\frac{NA_i(OVB)}{NA_i(MVB)} \approx \frac{1 + E_{Oi}(q_t - 1)}{1 + E_{Mi}(q_t - 1)} \cdot C(q_k), \tag{3.31}$$

where $C(q_k)$ is a function of q_k, and E_{Oi} and E_{Mi} correspond to the evolution rates at level i for the OVB- and MVB-tree, respectively. Given that typically E_{Oi} is an order of magnitude larger than E_{Mi}, the ratio in the above equation initially increases with q_t (for small q_t and fixed q_k) and will converge to $E_{Oi} \cdot C(q_k)/E_{Mi}$ when q_t is sufficiently large. These observations are experimentally evaluated in Section 5.

4. GENERALIZATION OF THE ANALYSIS FRAMEWORK

The models of the previous section assume that no buffer is available, and the datasets are uniform whereas the agility remains constant for each timestamp. Section 4.1 extends the cost models to estimate page accesses in the presence of LRU buffers, and Section 4.2 addresses general data distributions and variable agilities. Finally, Section 4.3 applies the entire analysis framework to R-trees.

4.1 Introducing a LRU Buffer

Equations (3.16) and (3.23) estimate the number of node accesses of OVB- and MVB-trees. In practical environments, buffers are widely used to improve query efficiency, and ignoring the effect of buffering may bias performance evaluation. Therefore, it is important for the proposed models to capture the number of disk page accesses when a (typically, LRU) buffer is available.

The behavior of LRU buffers in general database environments has been analyzed in Bhide et al. [1993]. Subsequent work [Huang et al. 1997; Leutenegger and Lopez 2000] that addresses the buffered performance of specific index structures is based on Bhide et al.'s [1993] findings that, after the buffer warmup period (i.e., all buffer pages have been loaded), the probability of locating a page in the buffer does not vary significantly. Based on this idea, Leutenegger and

Lopez [2000] adopt a two-step approach to predict the I/O of access methods. Assuming an empty buffer with C pages, the first step aims at estimating the number n_q of queries performed when all C pages are loaded. Let $prob(s_i, q)$ denote the probability that a node at level i is accessed by a query; then the probability that a level i node is accessed at least once by any of the n_q queries is: $1-[1-prob(s_i, q)]^{n_q}$. Since the total number of distinct pages accessed after n_q queries is C (recall that at this point the buffer becomes full for the first time), n_q can be solved from the following equation (K_i is the total number of nodes at level i).

$$\sum_{i=0}^{\lceil \log_f N/b \rceil} \left(K_i \cdot \left\{1-[1-prob\,(s_i, q)]^{n_q}\right\}\right) = C. \tag{4.1}$$

After obtaining n_q (which in practice can be saved in the system log), the second step estimates the number of page accesses for answering query q by observing that a node needs to be fetched from the disk if and only if the following conditions are satisfied: (1) the extents of the node intersect those of the query (for which the probability is $prob(s_i, q)$ for a node at level i), and (2) the node is not in the buffer. According to Bhide et al. [1993], the probability of (2) approximates the probability that the node is not accessed by any of the n_q queries during the buffer warmup period. Since this probability is $[1-prob(s_i, q)]^{n_q}$ for a level i node, the probability for the node to be loaded from the disk (to answer query q) is given by $prob(s_i, q)\cdot(1-prob(s_i, q))^{n_q}$. Therefore, the number of disk page accesses $PA(q)$ is given by

$$PA(q) = \sum_{i=0}^{\lceil \log_f N/b \rceil} \left\{K_i \cdot prob(s_i, q)\cdot[1-prob(s_i, q)]^{n_q}\right\}. \tag{4.2}$$

Finally, K_i and $prob(s_i, q)$ for OVB- and MVB-trees have already been discussed in the last section. Specifically, K_i is given in Equations (3.13) and (3.20), while for $prob(s_i, q)$:

$$\begin{aligned} OVB\_prob(s_i, q) &= \left(\frac{f^{i+1}}{N} + q_k\right) \frac{1+E_i\,(q_t-1)}{1+E_i\,(T-1)} \\ &= \left(\frac{f^{i+1}}{N} + q_k\right) \frac{1+\left[1-\left(1-\dfrac{f^{i+1}}{N}\right)^{2aN}\right](q_t-1)}{1+\left[1-\left(1-\dfrac{f^{i+1}}{N}\right)^{2aN}\right](T-1)}; \end{aligned} \tag{4.3}$$

$$\begin{aligned} MVB\_prob(s_i, q) &= \left(\frac{f_1^{i+1}}{N} + q_k\right) \frac{1+E_i(q_t-1)}{1+E_i(T-1)} \\ &= \left(\frac{f_1^{i+1}}{N} + q_k\right) \left[\frac{1+\dfrac{af_1^{i+1}\cdot(q_t-1)}{(b-f_1)^{i+1}}}{1+\dfrac{af_1^{i+1}\cdot(T-1)}{(b-f_1)^{i+1}}}\right]. \end{aligned} \tag{4.4}$$

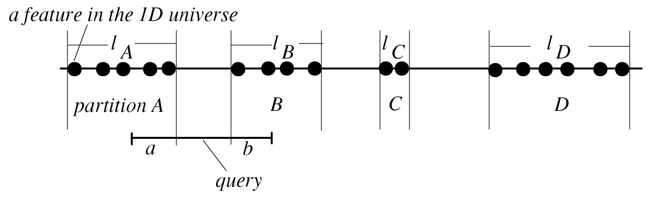


Fig. 9. A histogram example.

Applying the above equations in (4.1) and (4.2), we obtain cost models that predict the number of page accesses in answering a query with OVB- and MVB-trees when the buffer contains C pages.

4.2 General Temporal Datasets

In order to capture general temporal datasets, we need to allow arbitrary distributions and variable agilities for each timestamp. In practice, arbitrary data distributions are usually described using histograms, the idea of which is to divide the feature universe into a set of partitions such that the feature distribution of the objects within each partition is (almost) uniform [Piatetsky-Shapiro and Connell 1984; Liption et al. 1990]. Figure 9 illustrates an example where four partitions A, B, C, D are allocated. Each partition is associated with certain statistical information: typically, the number of features in the partition and the partition length. For example, there are five features in A, whose length is l_A. The histogram is used to estimate the selectivity of a query, or equivalently, the number of features to be retrieved. Consider, for example, the query in Figure 9 whose feature range intersects partitions A and B (with intersection lengths a and b). Given that A and B contain 5 and 4 features, respectively, the number of features retrieved can be estimated as: $5 \cdot (a/l_A) + 4 \cdot (b/l_B)$.

The description for data agility a_i, on the other hand, is straightforward: we only need to store a sequence of numbers representing the values of a_i at each timestamp. Observe that although values of a_i may differ significantly, consecutive $DIST_i$ are usually very similar (e.g., the balance distribution of accounts changes slowly from one timestamp to the next one). Since most queries in practice involve short intervals [Salzberg and Tsotras 1999], it is reasonable to consider that $DIST_i$ remains approximately the same at all timestamps i during q_t, which indicates that logical trees at these timestamps have similar structures.

Next we extend the analysis of Section 3 to address general temporal datasets. Without loss of generality, we denote the set of query timestamps as $t_1, t_2, \ldots, t_{qt}$. Let N_ε be the number of features retrieved at the first timestamp t_1. Since $DIST_i$ does not vary significantly[5] during q_t, approximately the same number of features is retrieved at every timestamp t_i ($1 \le i \le q_t$). Let S_1 be the set of nodes that are accessed at t_1 (i.e., these nodes are alive at t_1 and their

[5]In some extreme cases where distributions change drastically in consecutive timestamps, we can use the average number of features retrieved at each timestamp to replace N_ε.

feature ranges intersect q_k). Because logical trees at close timestamps have similar structures, nodes that evolve from those in S_1 at subsequent timestamps have similar feature ranges. As a result, the complete set of nodes accessed by a query is approximately the union of S_1 and the set S_E of nodes evolving from S_1 during q_t, or more formally, $NA(q) = |S_1| + |S_E|$. If $N_{\varepsilon i}$ is the number of level i nodes in S_1, then:

$$|S_1| = \sum_{i=0}^{h-1} N_{\varepsilon i}.$$

Given N_ε (obtained from histograms), the values for $N_{\varepsilon i}$ can be computed by observing that a range query on B-trees must visit a set of nodes whose feature ranges are consecutive. Since on average f entries are reported in each leaf node (with possible exceptions for the two nodes whose feature ranges include the endpoints of the query range), the number of expected leaf nodes accessed is $[N_\varepsilon/f]$. Following the same reasoning: $N_{\varepsilon i} = \max\{1, [N_\varepsilon/f^{i+1}]\}$ (the max operation is due to the fact that at least one node must be visited at each level even if no record is retrieved; that is, $N_\varepsilon = 0$).

Next we focus on $|S_E|$, for which we need to estimate the evolution rate $E_{\varepsilon i}$ for level i nodes in S_1. Unlike E_i in Section 3, $E_{\varepsilon i}$ is a function of time (represented as $E_{\varepsilon i}(j)$ for timestamp j) because the number of updates at each timestamp is different. Then the number of level i nodes that evolve from S_1 during q_t is given by $N_{\varepsilon i} \sum_{j=t_2}^{t_{qt}} E_{\varepsilon i}(j)$. So the total number of nodes in S_E is $\sum_{i=0}^{h-1} [N_{\varepsilon i} \sum_{j=t_2}^{t_{qt}} E_{\varepsilon_i}(j)]$. As a result,

$$NA(q) = |S_1| + |S_E| = \sum_{i=0}^{h-1} \left\{ N_{\varepsilon i} \left[1 + \sum_{j=t_2}^{t_{qt}} E_{\varepsilon_i}(j) \right] \right\}.$$

The derivation of $E_{\varepsilon i}$ follows that of E_i as presented in Section 3, except that, since we only consider a fraction of live nodes (specifically, $N_{\varepsilon i}$) at each timestamp, the number of updates that affect these nodes is $a_j \cdot N_{\varepsilon i} \cdot f^{i+1}$ (note that $N_{\varepsilon i} \cdot f^{i+1}$ is the number of entries in the subtrees of the $N_{\varepsilon i}$ nodes). Therefore, for OVB-trees,

$$E_{\varepsilon_i}(j) = 1 - \left(1 - \frac{1}{N_{\varepsilon i}}\right)^{2a_j \cdot N_{\varepsilon i} \cdot f^{i+1}}. \tag{4.5}$$

The new cost model that estimates the number of node accesses for OVB-trees is

$$OVB\_NA(q) = \sum_{i=0}^{\lceil \log_f N/b \rceil} \left(N_{\varepsilon i} \left\{ 1 + \sum_{j=t_2}^{t_{qt}} \left[1 - \left(1 - \frac{1}{N_{\varepsilon i}}\right)^{2a_j \cdot N_{\varepsilon i} \cdot f^{i+1}} \right] \right\} \right). \tag{4.6}$$

Similarly, for MVB-trees, the evolution rate $E_{\varepsilon i}(j)$ at timestamp j is given by

$$E_{\varepsilon_i}(j) = \frac{a_j \cdot f_1^{i+1}}{(b - f_1)^{i+1}}. \tag{4.7}$$

The corresponding formula for MVB-trees is

$$MVB\_NA(q) = \sum_{i=0}^{\lceil \log f_1 N/b \rceil} \left\{ N_{\varepsilon i} \left[1 + \frac{f_1^{i+1} \cdot \sum_{j=t_2}^{j=t_{qt-1}} a_j}{(b - f_1)^{i+1}} \right] \right\}. \quad (4.8)$$

The size estimation of OVB- and MVB-trees is the same as the uniform case, except that agility a_i (hence, E_i) may vary at each timestamp. The estimations for $E_i(j)$ (i.e., the evolution rate of level i at timestamp j) is similar to Equations (4.5) and (4.7), but the number of updates at each timestamp is $a \cdot N$. Replacing $E_i \cdot (T-1)$ with $\sum_{j=2}^{T} E_i(j)$in Equations (3.14) and (3.21), we obtain:

$$Size(OVB) = \sum_{i=0}^{\lceil \log_f N/b \rceil} \left(\frac{N}{f^{i+1}} \left\{ 1 + \sum_{i=2}^{T-1} \left[1 - \left(1 - \frac{f^{i+1}}{N} \right)^{2a_i N} \right] \right\} \right); \quad (4.9)$$

$$Size(MVB) = \sum_{i=0}^{\lceil \log f_1 N/b \rceil} \left[\frac{N}{f_1^{i+1}} + \frac{N \cdot \sum_{i=2}^{T-1} a_i}{(b - f_1)^{i+1}} \right]. \quad (4.10)$$

Following the analysis of Section 3.4, the estimation for selectivity becomes:

$$NUM(q) = N_\varepsilon \cdot \left(1 + \sum_{i=t_2}^{t_{qt}} a_i \right), \quad \text{and} \quad sel(q) = \frac{N_\varepsilon \cdot \left(1 + \sum_{i=t_2}^{t_{qt}} a_i \right)}{N \cdot \left(1 + \sum_{i=2}^{T-1} a_i \right)}. \quad (4.11)$$

Finally, we extend Equation (4.2) to capture the query performance when a LRU buffer is available. First observe that the expected number of disk page accesses $PA(q)$ can be written as

$$PA(q) = \sum_{i=0}^{h-1} \left\{ N_{\varepsilon i} \left[1 + \sum_{j=t_2}^{t_{qt}} E_{\varepsilon_i}(j) \right] \cdot \left[1 - prob_\varepsilon(s_i, q) \right]^{n_q} \right\}, \quad (4.12)$$

where $prob_\varepsilon(s_i, q)$ is the probability (averaged "locally," namely, among nodes in $|S_1|$ and $|S_E|$) that a node of level i is accessed by the query. The reasoning of the above equation is that $N_{\varepsilon i}[1 + \sum_{j=t_2}^{t_{qt}} E_{\varepsilon_i}(j)]$ is the number of nodes that must be accessed at level i, and $[1 - prob_\varepsilon(s_i, q)]^{nq}$ states the average probability that each of these nodes is not in the buffer (thus it must be fetched from the disk). We estimate $prob_\varepsilon(s_i, q)$ based on Equation (3.8) except that s_k and E_i are replaced with the corresponding locally averaged values. More specifically:

$$\overline{s_k} = (N_{\varepsilon i} \cdot f^{i+1}) \cdot \frac{q_k}{N_\varepsilon}, \quad \text{and} \quad \overline{E_i} = \frac{1}{q_t - 1} \sum_{j=t_2}^{t_{qt}} E_{\varepsilon_i}(j),$$

where $E_{\varepsilon i}$ is given in Equations (4.5) and (4.7), respectively, for OVB- and MVB-trees.

4.3 Extension to General MLTS

The previous analysis consists of two parts: $prob_{feature}$ and $prob_{tm}$ (see Equation (3.1). The properties of B-trees are only used for $prob_{feature}$, or more specifically, s_{ik} (the feature extent of a node). Extensions of the cost models to other MLTS differ only in the estimation of $prob_{feature}$, because general MLTS are constructed through the same overlapping or multiversion frameworks, and thus have similar temporal behavior. Therefore, provided that related results are available regarding the performance of an ephemeral structure, our framework produces models for the resulting MLTS easily. In the sequel, we illustrate this by considering the R-tree as the ephemeral structure; that is, the corresponding MLTS are overlapping R-trees (OVR-trees) and multiversion R-trees (MVR-trees).[6]

R-trees have been extensively used in spatial databases, where the universe is a two dimensional square. By Equation (3.2), $prob_{feature}$ should be represented as

$$prob_{feature} = (s_1 + q_1)(s_2 + q_2), \tag{4.13}$$

where q_1 and q_2 are the extents of q_k along the spatial dimensions.

The cost models for OVR- and MVR-trees differ from those for OVB- and MVB-trees only in the estimation for s_1 and s_2, which is the focus of R-tree analysis [Kamel and Faloutsos 1993; Pagel and Six 1996; Theodoridis and Sellis 1996; Theodoridis et al. 2000]. Particularly, when *DIST* is uniform, the estimation is given by Equations (4.14) and (4.15), where D_i refers to the *density* of node MBRs at level i (D_0 is the density D of the actual object MBRs).[7]

$$s_{i1} = s_{i2} = \sqrt{D_{i+1}\frac{f^{i+1}}{N}} \quad (0 \le i \le h-1) \tag{4.14}$$

where

$$D_{i+1} = \left(1 + \frac{\sqrt{D_i} - 1}{\sqrt{f}}\right)^2 \quad \text{and} \quad D_0 = D. \tag{4.15}$$

Replacing s_{ik} with s_{i1} and s_{i2} in Equations (3.16) and (3.23), we obtain the cost models for OVR- and MVR-trees as in Equations (4.16) and (4.17) respectively.

[6]As mentioned earlier, existing implementations of OVR- and MVR-trees are HR- [Nascimento and Silva 1998] and BTR-trees [Kumar et al. 1998], respectively.

[7]The density of a set of rectangles is defined as the average number of rectangles that contain a given point in the workspace. Equivalently, density can be expressed as the ratio of the sum of the areas of all rectangles over the area of the available workspace. Equations (4.14) and (4.15) assume that the MBRs of nodes in R-trees are quadratic, which is a widely accepted fact in R-tree cost analysis [Pagel and Six 1996; Theodoridis and Sellis 1996].

Similarly, (4.18) and (4.19) compute the tree sizes, and Equation (4.20) predicts the selectivity of interval queries. Note that the notions of degradation agility and multitree point apply to general MLTS by following the same reasoning as for B-trees.

$$
\begin{aligned}
\mathit{OVR\_NA}(q) &= \sum_{i=0}^{\lceil \log_f N \rceil - 1} \left(\frac{N}{f^{i+1}} \left(\sqrt{D_{i+1} \frac{f^{i+1}}{N}} + q_1 \right) \left(\sqrt{D_{i+1} \frac{f^{i+1}}{N}} + q_2 \right) \right. \\
&\qquad \left. \times \left\{ 1 + \left[1 - \left(1 - \frac{f^{i+1}}{N_\varepsilon} \right)^{2a_j \cdot N_\varepsilon} \right] (q_t - 1) \right\} \right). \qquad (4.16) \\
\mathit{MVR\_NA}(q) &= \sum_{i=0}^{\lceil \log_{f_1} N \rceil - 1} \left\{ \frac{N}{f_1^{i+1}} \left(\sqrt{D_{i+1} \frac{f_1^{i+1}}{N}} + q_1 \right) \left(\sqrt{D_{i+1} \frac{f_1^{i+1}}{N}} + q_2 \right) \right. \\
&\qquad \left. \times \left[1 + \frac{a f_1^{i+1} \cdot (q_t - 1)}{(b - f_1)^{i+1}} \right] \right\}. \qquad (4.17) \\
\mathit{Size}(\mathit{OVR}) &= \sum_{i=0}^{\lceil \log_f N \rceil - 1} K_i = \sum_{i=0}^{h-1} \left\{ \frac{N}{f^{i+1}} + \left[1 - \left(1 - \frac{f^{i+1}}{N_\varepsilon} \right)^{2a_j \cdot N_\varepsilon} \right] \right. \\
&\qquad \left. \times \frac{N}{f^{i+1}} (T - 1) \right\}. \qquad (4.18) \\
\mathit{Size}(\mathit{MVR}) &= \sum_{i=0}^{\lceil \log f_1 N \rceil - 1} K_i = \sum_{i=0}^{h-1} \left[\frac{N}{f_1^{i+1}} + \frac{a f_1^{i+1} N (T - 1)}{(b - f_1)^{i+1}} \right]. \qquad (4.19) \\
\mathit{sel}(q) &= \left(\sqrt{\frac{D}{N}} + q_1 \right) \left(\sqrt{\frac{D}{N}} + q_2 \right) \frac{1 + a(q_t - 1)}{1 + a(T - 1)}. \qquad (4.20)
\end{aligned}
$$

Extending the above models to address performance under buffers and arbitrary data distribution is also straightforward. Specifically, $\mathit{prob}(s_i, q)$ is estimated as Equations (4.21) and (4.22), respectively, for OVR- and MVR-trees. Replacing $\mathit{prob}(s_i, q)$ in Equation (4.2) with these two equations we obtain models that predict the number of page accesses.

$$
\begin{aligned}
\mathit{OVR\_prob}(s_i, q) &= \left(\sqrt{D_{i+1} \frac{f^{i+1}}{N}} + q_1 \right) \left(\sqrt{D_{i+1} \frac{f^{i+1}}{N}} + q_2 \right) \\
&\qquad \times \frac{1 + \left[1 - \left(1 - \frac{f^{i+1}}{N} \right)^{2aN} \right] (q_t - 1)}{1 + \left[1 - \left(1 - \frac{f^{i+1}}{N} \right)^{2aN} \right] (T - 1)}. \qquad (4.21)
\end{aligned}
$$

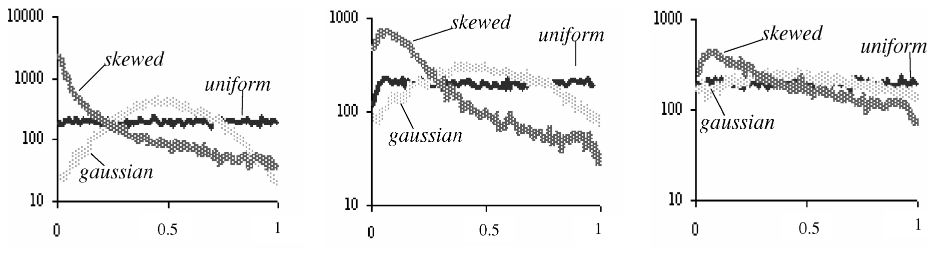
$$OVR\_prob(s_i, q) = \left(\sqrt{D_{i+1}\frac{f_1^{i+1}}{N}} + q_1\right)\left(\sqrt{D_{i+1}\frac{f_1^{i+1}}{N}} + q_2\right) \times \left[\frac{1 + \frac{af_1^{i+1}\cdot(q_t - 1)}{(b - f_1)^{i+1}}}{1 + \frac{af_1^{i+1}\cdot(T - 1)}{(b - f_1)^{i+1}}}\right]. \quad (4.22)$$

Arbitrary data distribution (particularly query selectivity) can be accurately captured by extending the histogram methods for 1-D features. By applying the analysis in Section 4.1, we obtain equations that have exactly the same form as Equations (4.6) and (4.8) through (4.12) for query performance and tree size of OVR- and MVR-trees, as well as the query selectivity. The difference is in the way that values of N_ε and N_{ei} are obtained, which is the topic of the analysis of R-trees [Acharya et al. 1999; Theodoridis et al. 2000].

Our cost models assume that MLTS are searched in a recursive depth-first manner. That is, if all the branches of a node have been visited, we backtrack to the parent node and continue with the next qualifying nontraversed branch. In spite of the fact that, as mentioned earlier, specialized algorithms (i.e., backward pointers [Bercken and Seeger 1996]) may improve the performance of interval queries, we chose to present our model using the recursive depth-first algorithm for the following reasons. First, the backward pointer method applies only to a few ephemeral structures. For others (e.g., R-tree based structures), it is necessary to visit qualifying entries at all levels. Since we aim at the general methodology, we based our derivation on the recursive algorithm. Second, it is straightforward to adapt the proposed models to the backward pointer algorithm: the estimation for the leaf nodes remains the same, whereas we only need to set q_t to 1 to obtain estimations for the other levels.

5. EXPERIMENTAL EVALUATION

In this section, we conduct an extensive experimental evaluation to prove the efficiency of the proposed models, first discussing MLTS based on B-trees. Due to the lack of real datasets, we created synthetic ones as follows. At the first timestamp, the features (each feature is a single value) of 20K objects distribute in the universe [0, 1] following uniform, skewed (i.e., Zipf with harmonic constant 0.6), or Gaussian (with mean 0.5, variance 0.2) distributions. Then, at each of the following 199 timestamps (i.e., the history contains 200 timestamps), $a\%$ of the objects are selected to produce feature changes (a ranges from 0 to 100%). For example, if a equals 25%, 5K (= 25% × 20K) objects issue updates per timestamp and the entire dataset contains 1 million (= 5K × 200) records. Each feature update is such that the feature of an object deviates from its previous value by a distance randomly generated in [−0.05, 0.05]. In this way, we allow the distribution of the objects' features to vary slowly along with time. Figure 10 shows the histograms at timestamps 1, 100, and 200 for datasets with the same agility $a = 10\%$ but different initial distribution $DIST_1$. Notice that as time evolves the data distribution gradually becomes uniform. In order



(a) (b) (c)

Fig. 10. Histograms of temporal datasets: (a) timestamp 1; (b) timestamp 100; (c) timestamp 200.

to obtain the histogram at each timestamp, we divide the feature universe into 100 partitions with equal lengths, and store, for each partition, the number of objects in the corresponding range.

Variable agilities were created by setting a value for a_{MAX} and randomly generating agilities in the range $[0, a_{MAX}]$ for each timestamp. In the sequel we denote a dataset with respect to its initial distribution and agility characteristics. For example, $Uni_{10\%}$ refers to uniform distribution and fixed agility 10%, and $Sk_{25\%R}$ represents a dataset with skewed initial distribution and randomized agility (note the "R" behind the agility value) generated by $a_{MAX} = 25\%$. Similarly, Gaussian distribution is denoted as *Gau*.

Query performance is measured by the average node (or page) accesses in answering a workload consisting of 500 queries. All the queries in a workload involve a feature range of the same length q_k and an interval range with the same number of timestamps q_t. The left endpoints of the feature and time ranges of query q are uniformly distributed in ranges $[0, 1-q_k]$, and $[1, 201-q_t]$, respectively. In the sequel, we denote a workload as $WRK_{qk,qt}$ to indicate its parameters. OVB- and MVB-trees are implemented as described in Salzberg and Tsotras [1999] and Becker et al. [1996], respectively. Unless otherwise stated, the parameters for MVB-trees are: $P_{SVO} = 0.8$ (suggested value in Varman and Verma [1997]) and $P_{SVU} = P_{SVO}/2$, $P_U = P_{SVO}/4$. The page size is set to 1 K or 4 K bytes resulting in node capacities 122 or 506 entries, respectively, for OVB-trees. The corresponding numbers for MVB-trees are 61 and 253 entries. These values are used by the cost models.

5.1 Structure Size

We first evaluate Equations (3.14), (3.21), (4.9), and (4.10) on the sizes of the MLTS. Figure 11 shows the estimated (*est*) and experimental (*exp*) sizes for uniform datasets as a function of dataset agility a, which remains fixed throughout history. Notice that OVB-trees initially grow very fast with a, but their sizes stabilize after the degradation agility (around 4 and 1% for 1 and 4 K block sizes, respectively), where they degenerate into independent trees. The degradation agility is lower for 4 K blocks because larger nodes are more likely to be duplicated at each timestamp. On the other hand, the sizes of MVB-trees grow linearly with the dataset agility and are much more space efficient than OVB-trees for usual agilities (up to 25%).

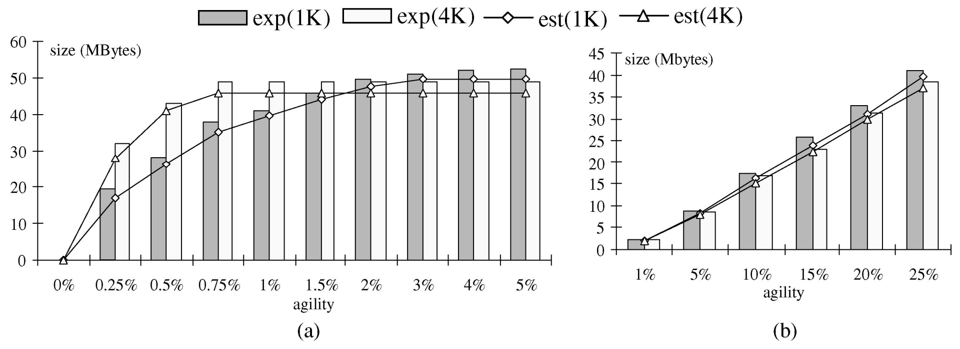


Fig. 11. Sizes of MLTS as a function of agility (uniform datasets): (a) OVB-trees; (b) MVB-trees.

Table II. Comparison of Tree Sizes

Sizes (Mbytes) of OVB-trees (fixed agility)								
Agility (%)	Uniform		Skewed		Gaussian		Estimated	
	1 K	4 K	1 K	4 K	1 K	4 K	1 K	4 K
0.25	19	32	17	31	17	31	17	29
0.5	28	43	27	43	27	42	28	41
0.75	38	49	37	49	37	48	35	46
1	41	49	41	49	40	49	39	46
2	50	49	49	49	50	49	48	46
3	51	49	51	49	51	49	49	46
4	52	49	52	49	51	49	49	46
5	52	49	52	49	51	49	49	46

Sizes (Mbytes) of MVB-trees (random agility)								
Random Agility (%)	Uniform		Skewed		Gaussian		Estimated	
a_{MAX}	1 K	4 K	1 K	4 K	1 K	4 K	1 K	4 K
1	1	1	1	1	1	1	1	1
5	4	4	4	4	4	4	4	4
10	9	8	8	8	9	8	8	8
15	12	11	12	11	11	11	10	9
20	16	15	16	15	15	14	14	13
25	20	19	21	19	20	19	18	17

In order to verify the estimated degradation agility for MVB-trees (81%), we increased the agility 5% at a time while checking for any noticeable increase in the tree sizes. We found that the sizes of MVB-trees (for both block sizes) stabilized at around 120 megabytes when the agility became higher than 85%. Above $a = 35\%$, MVB-trees consumed more space than independent B-trees (i.e., around 50 Mb as shown in Figure (11a)), which is consistent with the estimated value of size-MTP (33%) obtained by Equation (3.29).

Table II shows in detail the tree sizes for datasets of all distributions under various agilities. The estimations are the same for datasets with the same agility, and the experimental values confirm that the size of a tree is hardly affected by the data distribution. Note that although we have shown only the

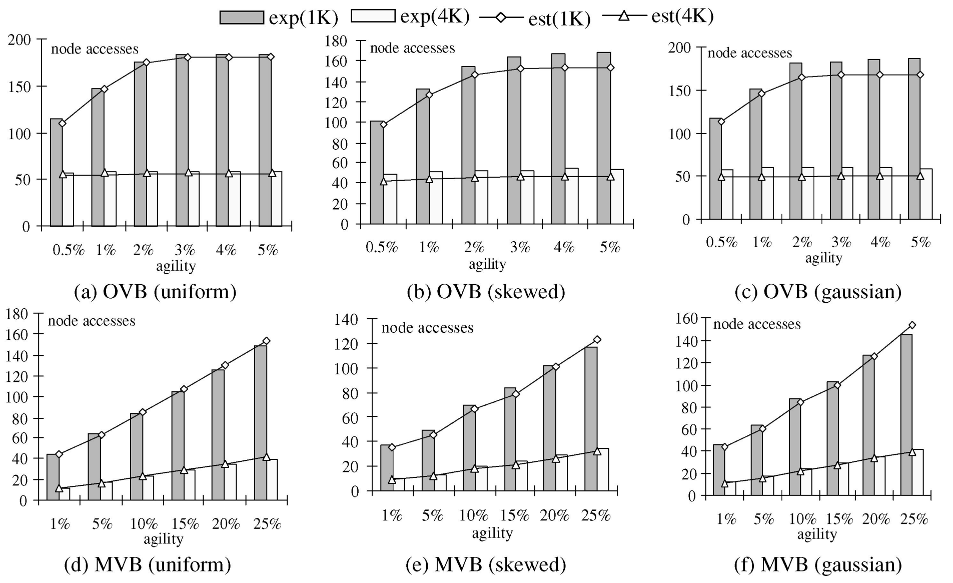


Fig. 12. Node accesses versus agility ($q_k = 6\%, q_t = 10$).

sizes of OVB- (MVB-) trees for fixed (random) agilities to avoid redundancy, sizes of the other trees are also very well predicted (with similar error rates).

The small error (below 10%) in all cases (also in subsequent results) is due to underestimation of the actual values. This is because we only consider the major structural changes (i.e., node duplication and version splits for OVB- and MVB-trees), while omitting other types of changes that may increase the tree size (e.g., key splits or strong version overflows in OVB- and MVB-trees may create two new nodes at a time). As mentioned in Section 3.1 and confirmed in our experiments, however, such changes occur very infrequently, and their omission does not bias the estimated performance significantly.

5.2 Query Performance

Next we evaluate query performance starting with the case where no buffer is available. We identify several parameters that affect performance: the data agility a, the feature range q_k, and the query length q_t. In order to investigate the effects of individual parameters, in each experiment we vary one parameter and fix the others to some standard values: $a = 2\%$ for OVB- and 10% for MVB-trees, $q_k = 6\%$ of the entire feature universe and $q_t = 10$ timestamps (5% of the history). Note that the standard agility is lower for OVB-trees because their degradation agilities are lower.

5.2.1 *Performance Without Buffers.* Node accesses for uniform datasets are predicted using Equations (3.16) and (3.23), whereas Equations (4.6) and (4.8) together with histograms (as shown in Figure 10) are used for nonuniform datasets. For the first experiment we fix q_k and q_t, and vary the agility of the datasets (which, however, remains constant for each timestamp). Figure 12

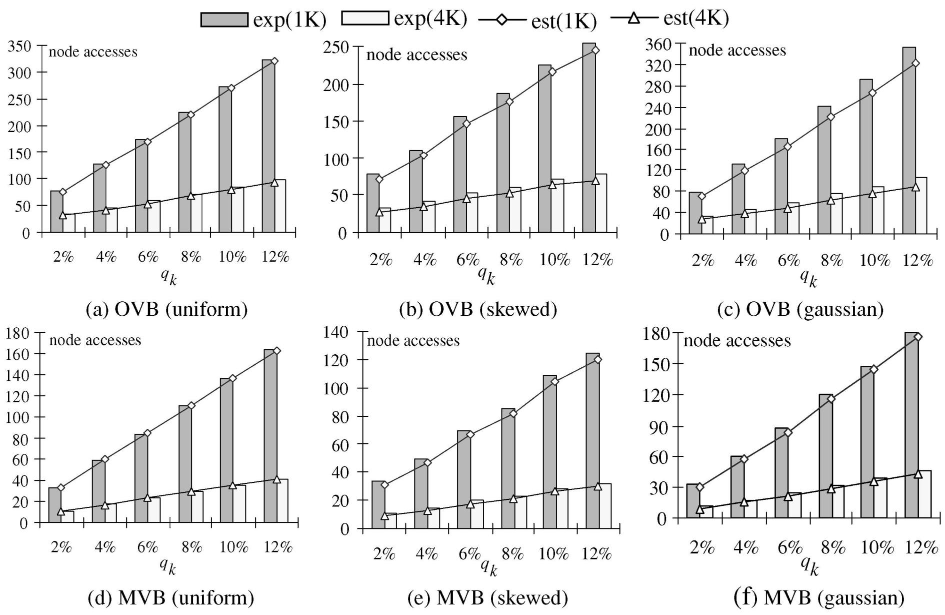


Fig. 13. Node accesses versus q_k ($a = 2\%$ (OVB) or 10% (MVB), $q_t = 10$).

shows the node accesses (NA) as a function of agility for datasets with different distributions using workload $WRK_{6\%,10}$. The performance of both structures demonstrates similar behavior to that of their sizes. Specifically, the query cost of OVB-trees grows quickly with agility (until the degradation agility is reached) and that of MVB-trees increases linearly. Note that the costs of OVB-trees with 4 K block sizes are constant because at agility 2% these trees have already degraded. Furthermore, by comparing figures on the same columns, it is clear that MVB-trees outperform OVB-trees significantly. For example, when agility = 5%, MVB-trees answer the query with around 60 (1 K block size) node accesses whereas OVB-trees (already degraded) must perform more than 150 node accesses. Notice that although for nonuniform data we report the average costs, for each individual query[8] we observe similar error rates to those in the average case. The same is true for all experiments.

Next we fix agility to 2 and 10% for OVB- and MVB-trees, respectively, q_t to 10 timestamps, and investigate query performance as q_k changes from 2 to 12% of the feature universe. Figure 13 shows the performance as a function of q_k. As expected, the costs of all trees increase linearly with q_k. MVB-trees outperform OVB-trees significantly (around 50%) in all cases. The estimated and experimental results are again very similar.

Figure 14 illustrates the node accesses for workloads $WRK_{6\%,1\sim25}$, (i.e., q_s is fixed to 6% and q_t ranges from 1 to 25 timestamps). OVB-trees outperform MVB-trees only on workloads with timestamp queries (q_t =1); for 1 and 4 K block size, the costs of OVB-trees are around 19 and 5 node accesses, respectively,

[8]The costs of queries on nonuniform data depend on concrete query locations, as captured by our model.

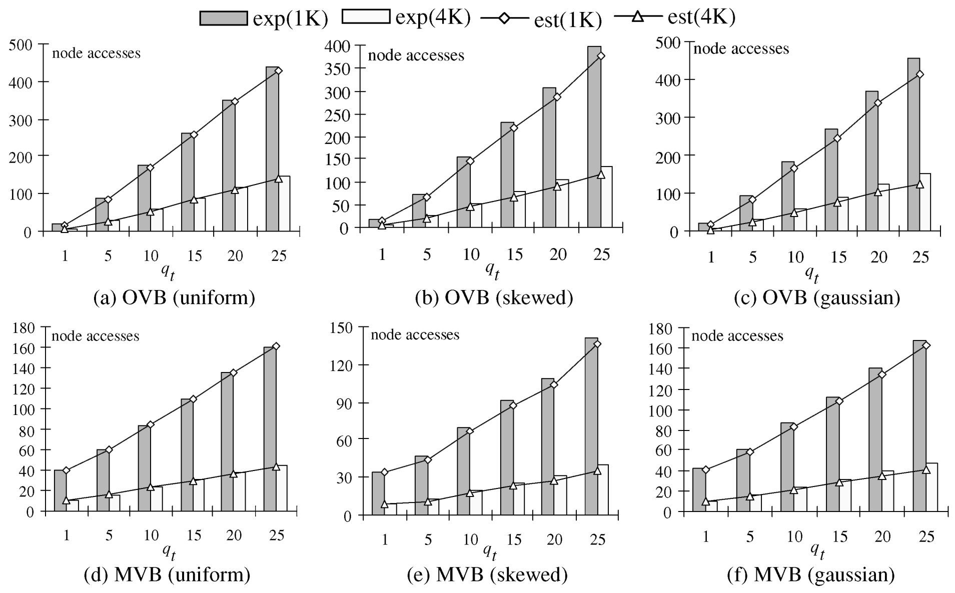


Fig. 14. Node accesses versus q_t ($a = 2\%$ (OVB) or 10% (MVB), $q_k = 6\%$).

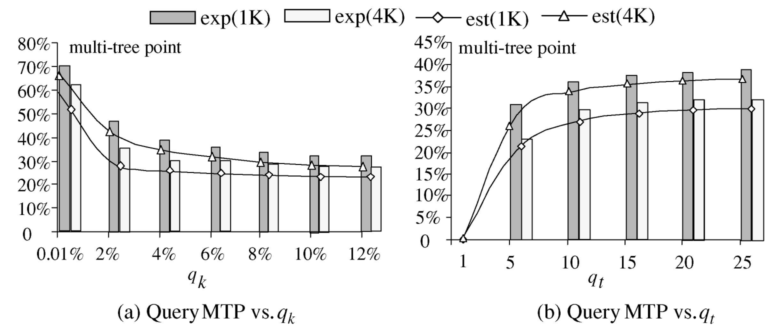


Fig. 15. Query multitree points (uniform).

and the corresponding MVB-tree costs are around 40 and 10. This is because for timestamp queries, only one logical B-tree needs to be visited and the fanout of nodes in OVB-trees is around twice that of MVB-trees, which accounts for the fact that OVB-trees require only half of the node accesses of MVB-trees. The performance of OVB-trees, however, deteriorates very quickly as q_t increases. The gain of MVB- over OVB-trees increases initially with the query interval, but tends to stabilize when q_t is longer than 10, which is consistent with Equation (3.31).

As discussed in Section 3.6, above a certain agility (i.e., the query MTP), MVB-trees will be outperformed by the independent tree implementation. In order to obtain the MTP for a set of specific query parameters (q_k, q_t) we increase the agility by 1% at each step, and then compare the performance of the corresponding MVB-trees with the independent tree implementation. Figures 15(a)

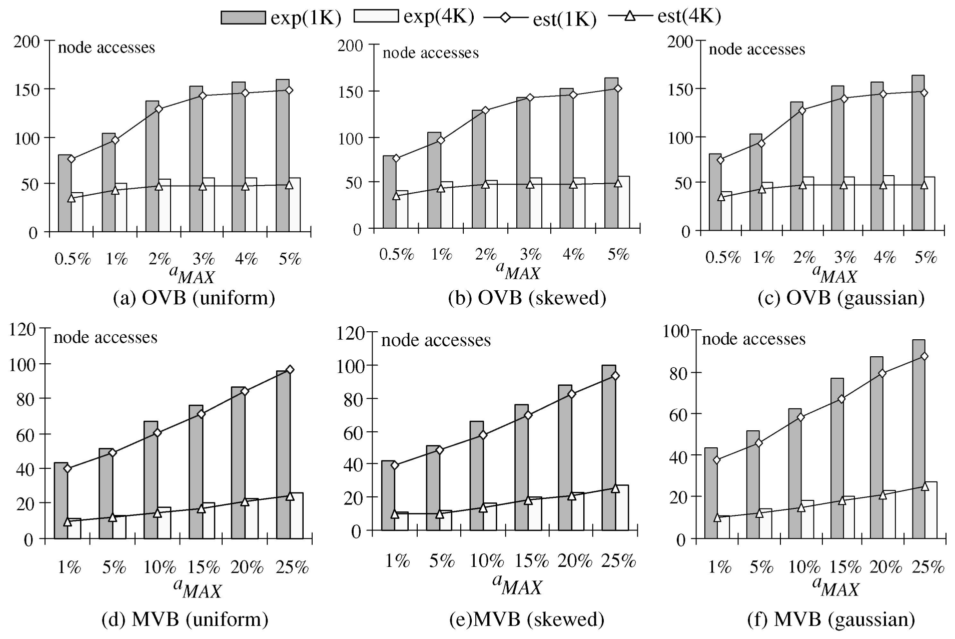


Fig. 16. Node accesses versus a_{MAX} ($q_k = 6\%, q_t = 10$).

and (b) demonstrate the estimated (by Equation (3.30) and experimental query MTP (for uniform datasets) with respect to q_k and q_t, respectively. As q_k (q_t) increases, the multitree point initially decreases (increases) very quickly and converges after around $q_k = 10\%(q_t = 15$ timestamps) to rather low values (30 and 35% for 1 and 4 K block sizes, respectively); above these agilities, the independent tree implementation (or OVB-trees) will outperform MVB-trees. Furthermore, recall (Section 5.1) that the size MTP is also around 35%, which indicates that MVB-trees are useful only for dataset agilities up to this percentage.

Figure 16 evaluates query performance for datasets with random agilities (node accesses plotted as a function of a_{MAX}), using workload $WRK_{6\%,10}$. The results with respect to other query parameters are omitted due to their similarity to those in Figures 13 and 14. The cost models are again very accurate.

5.2.2 *Performance with Buffers.* Now we proceed to evaluate the cost models (Equations (4.3), (4.4), and (4.12) in the presence of LRU buffers. Specifically, we repeat the experiments of Section 5.2.1 by introducing a LRU buffer with size 20% of the corresponding tree. Since OVB-trees are larger than MVB-trees, they require more buffer space to achieve the same percentage. Each workload now contains 2000 queries in order to allow a period long enough for the buffer to warm up. We start to measure the number of disk accesses after all the pages in the buffer are loaded for the first time (the longest warmup period requires up to 650 queries, meaning that at least 1350 queries are measured).

Because the results are very similar to those in the absence of the buffer, we selected only one diagram for datasets with variable agilities since they produce

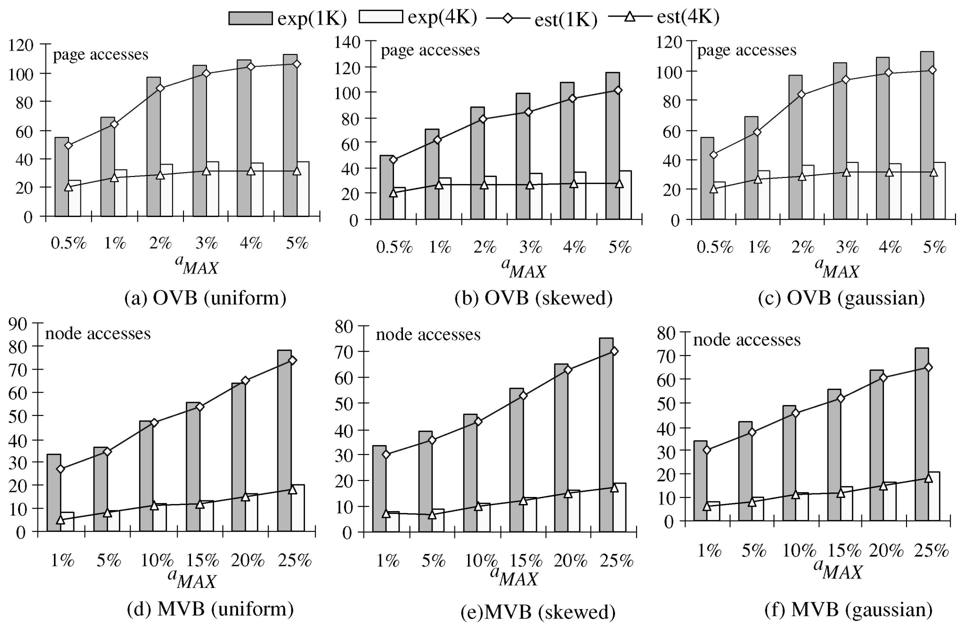


Fig. 17. Page accesses versus a_{MAX} ($q_k = 6\%, q_t = 10$).

relatively larger errors than datasets with fixed agilities. Figure 17 shows the number of page accesses as a function of a_{MAX} using workload $WRK_{6\%,10}$. The diagrams are similar to those in Figure 16, the only difference being that the buffer decreases the query costs, which are captured by our cost models very well.

In order to further study the buffer impact, we vary its size from 1 to 50% of the corresponding tree. Figure 18 shows the page accesses for workload $WRK_{6\%,10}$ as a function of the buffer size, using datasets with fixed agilities (2 or 10% for OVB- and MVB-trees, respectively). Both structures improve significantly only when the buffer contains a large part of the tree. For example, in our settings, a buffer that keeps 20% of an OVB- (MVB-) tree amounts to around 10 M (3 M) bytes. Given that practical applications usually involve much higher cardinalities and longer "active" history (i.e., timestamps that are subject to frequent queries), the required buffer size may be prohibitive. Therefore, in practice buffering may be of limited importance for MLTS.

5.3 Effects of P_{SVO}

In this section we demonstrate the effects of the P_{SVO} parameter on the performance MVB-trees using dataset $Uni_{10\%}$. Similar results were observed for other datasets. Figure 19 shows the tree sizes as a function of P_{SVO}. As discussed in Section 3.3, smaller P_{svo} lowers the tree sizes, indicating less data redundancy. Note that the MVB-tree achieves the smallest size (less than half of the largest size) at P_{svo} value 0.3, whereas the tree size starts to grow at smaller values, as predicted in Section 3.3.

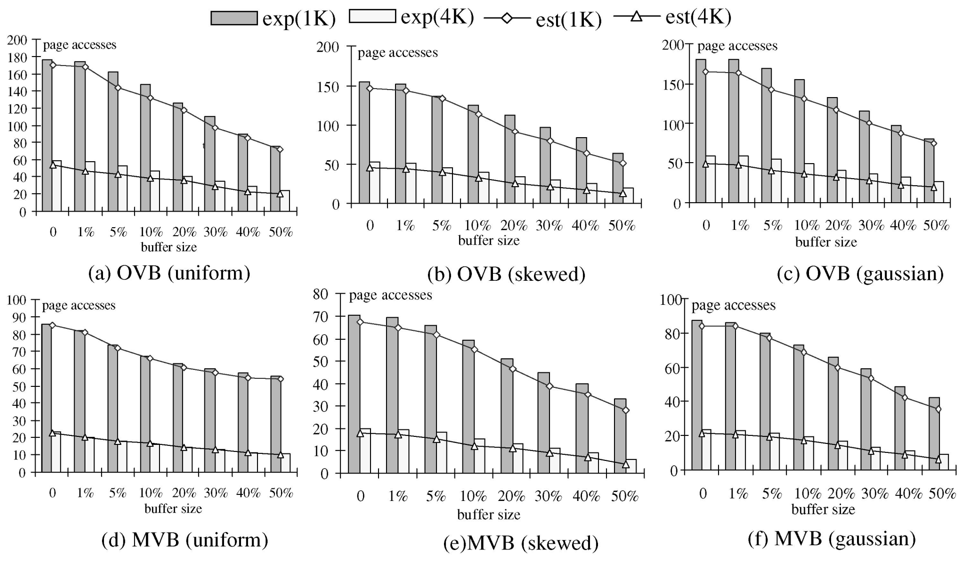


Fig. 18. Page accesses versus buffer size ($a = 2\%$ (OVB) or 10% (MVB), $q_k = 6\%, q_t = 10$).

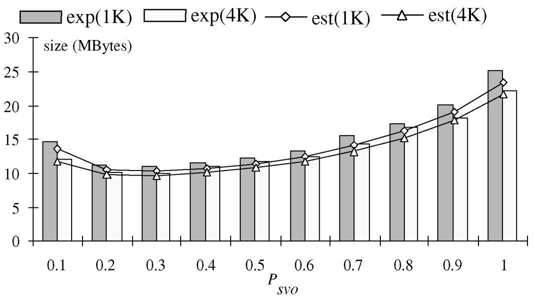


Fig. 19. Effects of P_{SVO} on the sizes of MVB-trees ($a = 10\%$).

Figure 20 shows the node and page accesses (buffer sizes fixed to 20% of the tree with $P_{SVO} = 0.8$) incurred by workload $WRK_{6\%,10}$ for trees built with different P_{SVO}. In this case, the optimal value of P_{SVO} (0.6) is different from the corresponding value for size minimization. Our cost models can lead to optimization heuristics that take into account both the space consumption and query performance. For example, if the objective is to minimize $NA(q)/Size(MVB)$, we can obtain the optimal P_{SVO} by solving:

$$\frac{d[NA(q)/Size(MVB)]}{df_1} \cdot \frac{df_1}{dP_{SVO}} = 0.$$

5.4 Selectivity Estimation

In this section we evaluate the efficiency of Equations (3.25) and (4.11) for estimating query selectivity. Figure 21 shows the actual and estimated number of records that satisfy the query conditions for datasets with skewed distribution

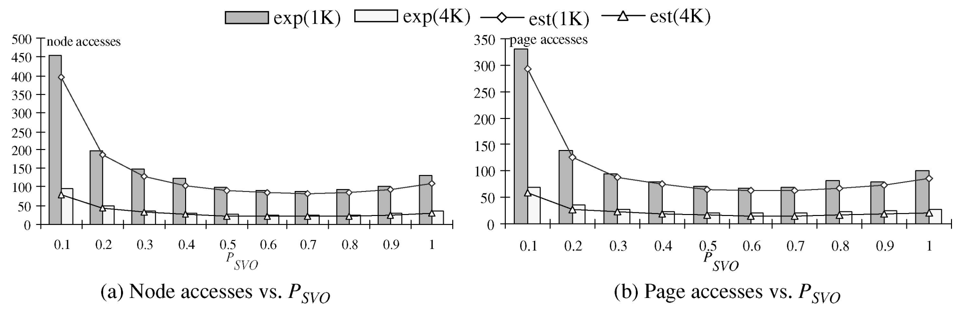


Fig. 20. Performance versus P_{SVO} ($a = 10\%, q_k = 6\%, q_t = 10$).

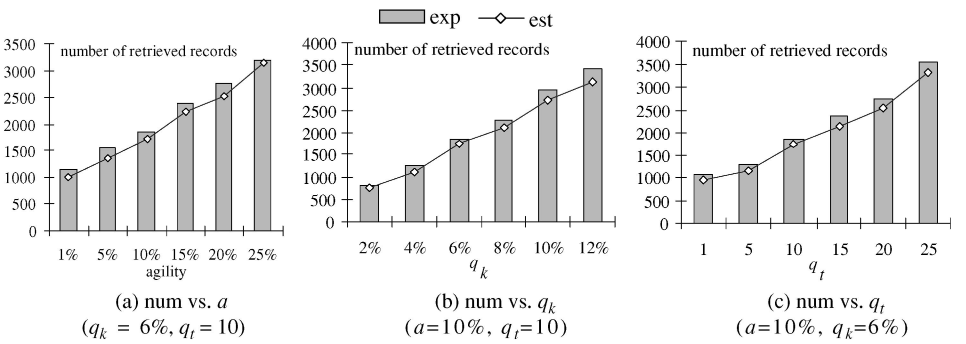


Fig. 21. Selectivity evaluation for skewed datasets with fixed agilities.

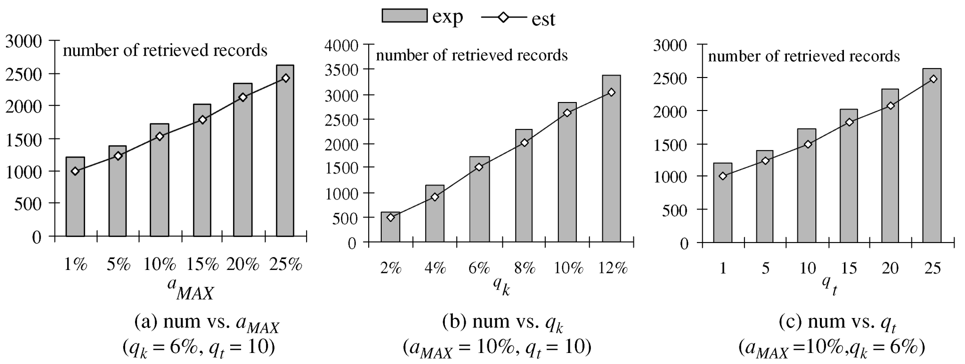


Fig. 22. Selectivity evaluation for skewed datasets with random agilities.

and fixed agilities. We omit the results for uniform and Gaussian distributions because the trends are similar (actually since skewed datasets usually produce the largest errors, the estimations are even more accurate for the other datasets). Figure 22 illustrates the results of similar experiments on skewed datasets with random agilities. It is clear that our models predict the selectivity accurately in all cases (with less than 10% errors).

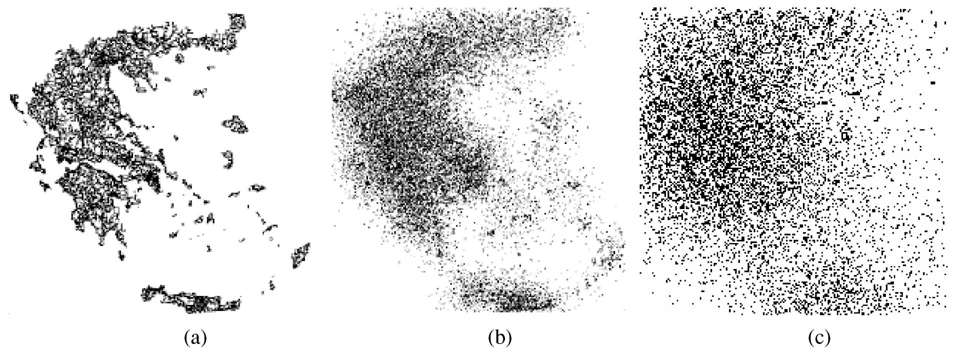


Fig. 23. Evolution of *GR* dataset at different timestamps: (a) 1; (b) 100; (c) 200.

5.5 Application to R-Trees

To demonstrate the generality of our approach, we evaluate the accuracy of the models proposed in Section 4 for MLTS based on R-trees. The OVR- and MVR-trees are implemented as HR- [Nascimento and Silva 1998] and BTR-trees Kumar et al. [1998]. The parameters of MVR-trees follow those in Kumar et al. [1998]: $P_U = 0.2$, $P_{SVO} = 0.85$, and $P_{SVU} = 0.4$. Using a page size of 1 K (4 K) bytes, the node capacities in OVR- and MVR-trees is 48 (202) and 34 (143) entries, respectively. Datasets are generated in a way similar to those used for OVB- and MVB-trees. Specifically, at the first timestamp objects are distributed in a 2-D unit universe by a certain distribution $DIST_1$. Then, at each of the subsequent 199 timestamps, $a\%$ (fixed or randomized) of the objects produce updates. An update causes the center of an object to move by (δx, δy), where δx and δy are randomly generated in $[-0.05, 0.05]$. We conducted experiments with uniform, skewed, and Gaussian distributions, and observed that the results were similar to those already reported in the previous sections. In order to avoid redundancy and further demonstrate the adaptability of our models, we generated datasets[9] by using a real-world map as $DIST_1$ of 23,268 objects [Web] (Figure 23(a)). The updates are generated in the same way as described earlier causing the initial distribution to gradually change. Figures 23(b) and (c) show examples of the dataset (referred to as *GR* in the sequel) at timestamps 100 and 200.

As before, in order to support nonuniform datasets, we maintain density maps [Theodoridis et al. 2000] at each timestamp. To be specific, we divide the universe into a grid with 50×50 cells and store in each cell the number of rectangles that intersect the cell (for a histogram) and the average density in the cell (for a density map) that is calculated as the ratio between the sum of the areas of the rectangles intersecting the cell and the cell's area.

Figures 24(a) and (b) evaluate the space consumption of OVR- and MVR-trees as a function of agility. The observations are similar to those for

[9]We are not aware of any real spatiotemporal datasets of sufficient size (in terms of objects and history length) available for experiments.

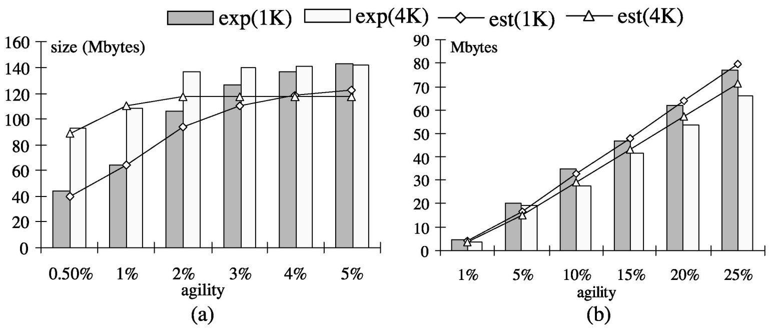


Fig. 24. Sizes of (a) OVR- and (b) MVR-trees.

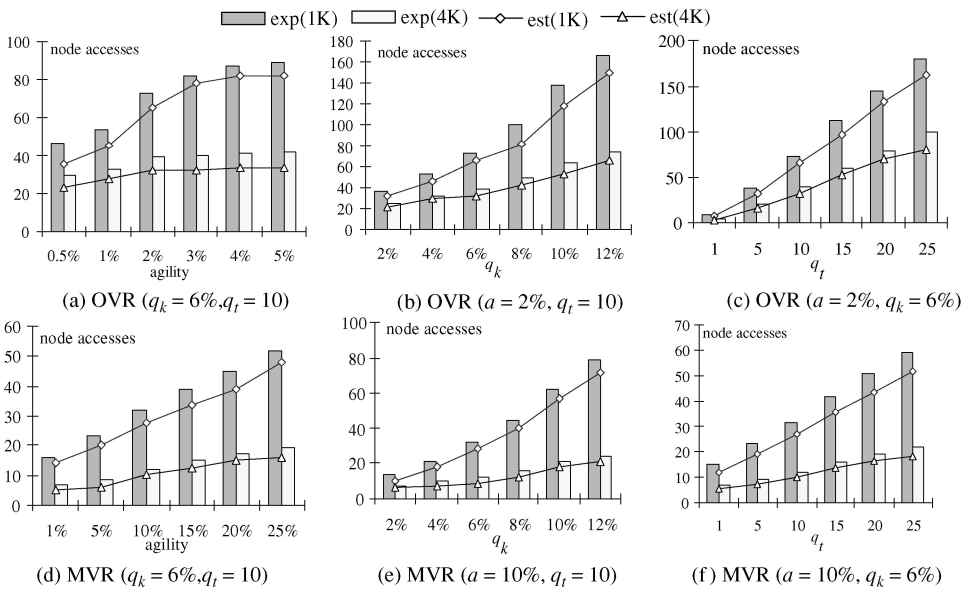


Fig. 25. Node accesses under fixed agilities.

OVB- and MVB-trees; OVR-trees degenerate into independent trees at very small agilities, whereas MVR-trees are more space efficient. Notice that (comparing with Figure 11) the degradation agilities of OVR-trees (for both page sizes) are slightly higher than those of OVB-trees due to the fact that the fanouts of nodes in OVR-trees are smaller.

Next we verify the predictions for the number of node accesses of OVR- and MVR-trees with respect to various dataset and query parameters. Figure 25(a) shows the number of node accesses as a function of agility for OVR-trees using workload $WRK_{6\%,10}$ (i.e., the feature of each query is a quadratic rectangle covering 0.36% of the entire universe). In Figures 25(b) and (c), the agility is fixed at 2%, and the performance is measured as q_k or q_t changes, respectively. Figures 25 (d) through (f) present the same experiments (except that the standard agility is set to 10%) for MVR-trees. The growth tendencies are similar to

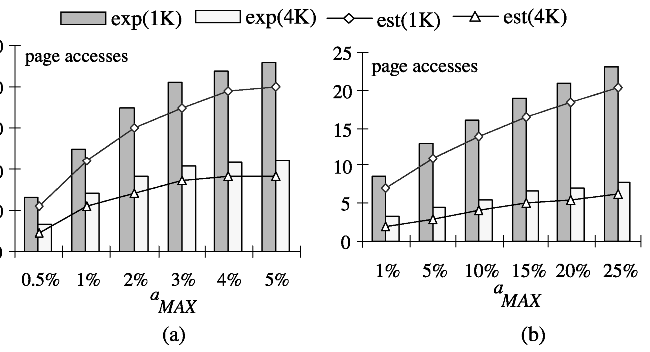


Fig. 26. Page accesses versus a_{MAX} ($q_k = 6\%, q_t = 10$): (a) OVB-trees; (b) MVB-trees.

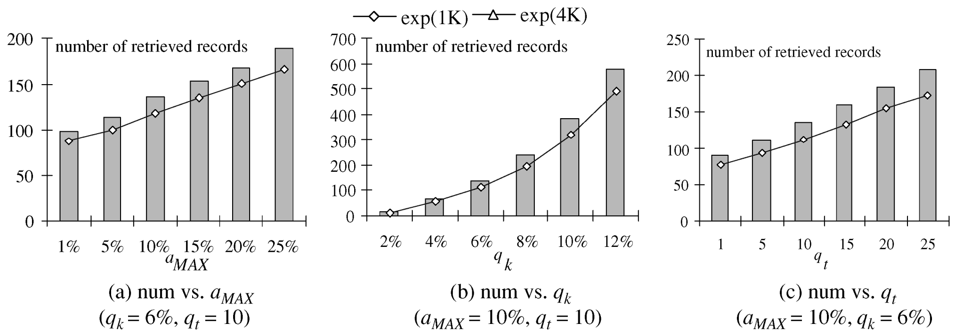


Fig. 27. Selectivity evaluation (for random agilities).

MLTS based on B-trees except that, in Figures 25(b) and (e), the cost increases quadratically with q_k due to the quadratic growth in the query feature area.

In Figure 26 we introduce a buffer with the size 20% of each tree, and measure the number of page accesses as a function of a_{MAX} (using workload $WRK_{6\%,10}$) for random agility datasets. The predicted costs (from Equation (4.12) follow the actual ones very well, yielding maximum error around 20%. Finally, Figure 27 evaluates Equation (4.11) which predicts query selectivity (again for datasets with random agilities).

To summarize, in this section we have presented an extensive experimental study on the accuracy and adaptability of our analysis. The results indicate that the proposed models capture the behavior of MLTS very well, producing accurate values for structure sizes, query performance (node or page accesses), and selectivity. Furthermore, we verified the observations discussed in Section 3.5 about the behavior of overlapping and multiversion structures (e.g., degradation agility, multitree points, etc.,). which provide strong indications towards selecting the best alternative access method in practice.

6. CONCLUSION

Our framework reduces the analysis of overlapping and multiversion structures to that of the corresponding ephemeral structures, which means that it is applicable to a variety of different access methods. Extensive experimentation proves the precision of the models for a wide range of conditions. To the best of our knowledge, this is the first work that attempts to provide a systematic study for these types of structures. Given the ever-increasing availability and importance of historical data in numerous applications, analysis of related structures is crucial for the development of efficient systems.

In addition to their usefulness for query optimization, the proposed models provide significant insight into the behavior of overlapping and multiversion structures. In particular, we establish, for the first time, the close connection between the performance of MLTS and the dataset agility. Furthermore, the formulae quantify how fast redundant data accumulate and predict the agilities where the structures degrade to independent trees. In general, multiversion structures usually incur less redundancy than overlapping structures and are preferable for general workloads. Above a certain agility (multitree points), however, the best alternative is to simply build an independent ephemeral structure for each timestamp. The proposed models accurately estimate the degradation agilities and the multitree points. Thus they can be employed by system administrators to decide suitable indexing methods given the dataset and system characteristics.

Furthermore, this article also lays down a solid foundation for investigating the performance of other queries, such as temporal and spatiotemporal joins [Soo et al. 1994; Zhang et al. 2002]; aggregate queries [Yang and Widom 2001; Papadias et al. 2002; Zhang et al 2001; Tao et al. 2002], and spatiotemporal nearest neighbor queries. It may also lead to the development of improved access methods. For example, structures such as MVB-trees were motivated by the need to optimize timestamp queries with minimal space overhead. Other structures could be developed aiming at the optimization of interval queries, balancing the tradeoff between data redundancy and performance.

REFERENCES

Acharya, S., Poosala, V., and Ramaswamy, S. 1999. Selectivity estimation in spatial databases. In *Proceedings of the ACM SIGMOD Conference* (June), 13–24.

Becker, B., Gschwind, S., Ohler, T., Seeger, B., and Widmayer, P. 1996. An asymptotically optimal multiversion B-tree. *VLDB J. 5*, 4, 264–275.

Bercken, J. V. D. and Seeger, B. 1996. Query processing techniques for multiversion access methods. In *Proceedings of the VLDB Conference* (Sept.), 168–179.

Bhide, A., Dan, A., and Dias, D. 1993. A simple analysis of LRU buffer replacement policy and its relationship to buffer warm-up transient. In *Proceedings of the International Conference of Data Engineering (ICDE)* (April), 125–133.

Burton, F. and Huntbach, M. 1985. Multiple generation text files using overlapping tree. *Comput. J. 28*, 4, 414–416.

Burton, F., Kollias, J., Kollias, V., and Matsakis, D. 1990. Implementation of overlapping B-trees for time and space efficient representation of collection of similar files. *Comput. J. 33*, 3, 279–280.

Carey, M., DeWitt, D., Richardson, J., and Shekita, E. 1986. Object and file management in the EXODUS extensible database system. In *Proceedings of the VLDB Conference* (August), 91–100.

Chien, S., Tsotras, V., Zaniolo, C., and Zhang, D. 2002. Efficient complex query support for multiversion XML documents. In *Proceedings of the Extending Database Technology Conference (EDBT)* (March), 161–178.

Easton, M. 1986. Key-sequence data sets on indelible storage. *IBM J. Res. Dev. 30*, 3, 230–241.

GARGANTINI, I. 1982. An efficient way to represent quadtrees. *Commun. ACM 25*, 12, 905–910.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference* (June), 47–57.

HUANG, Y., JING, N., AND RUNDENSTEINER, E. 1997. A cost model for estimating the performance of spatial joins using R-trees. In *Proceedings of the Scientific and Statistical Database Management Conference* (*SSDBM*) (August), 30–38.

JIANG, L., SALZBERG, B., LOMET, D., AND BARRENA, M. 2000. The BT-tree: A branched and temporal access method. In *Proceedings of the VLDB Conference* (Sept.), 451–460.

KAMEL, I. AND FALOUTSOS, C. 1993. On packing R-trees. In *Proceedings of the Conference on Information and Knowledge Management* (*CIKM*) (Nov.), 490–499.

KOLLIOS, G., GUNOPULOS, D., TSOTRAS, V., DELIS, A., AND HADJIELEFTHERIOU, M. 2001. Indexing animated objects using spatiotemporal access methods. *IEEE Trans. Knowl. Data Eng.* (to appear).

KUMAR, A., TSOTRAS, V., AND FALOUTSOS, C. 1998. Designing access methods for bitemporal databases. *IEEE Trans. Knowl. Data Eng. 10*, 1, 1–20.

LANKA, S. AND MAYS, E. 1991. Fully persistent B+-trees. In *Proceedings of the ACM SIGMOD Conference* (May), 426–435.

LEUTENEGGER, S. AND LOPEZ, M. 2000. The effect of buffering on the performance of R-trees. *IEEE Trans. Knowl. Data Eng. 12*, 1, 33–44.

LIPTION, R., NAUGHTON, J., AND SCHNEIDER, D. 1990. Practical selectivity estimation through adaptive sampling. In *Proceedings of the ACM SIGMOD Conference* (May), 1–11.

LOMET, D. AND SALZBERG, B. 1989. Access methods for multiversion data. In *Proceedings of the ACM SIGMOD Conference* (May), 315–324.

LOMET, D. AND SALZBERG, B. 1990. The performance of a multiversion access method. In *Proceedings of the ACM SIGMOD Conference* (May), 353–363.

NASCIMENTO, M. AND SILVA, J. 1998. Towards historical R-trees. In *Proceedings of the ACM Symposium on Applied Computing* (Feb.), 235–240.

PAGEL, B. AND SIX, H. 1996. Are window queries representative for arbitrary range queries? In *Proceedings of the ACM Symposium on Principles of Database Systems* (*PODS*) (June), 150–160.

PAPADIAS, D., TAO, Y., KALNIS, P., AND ZHANG, J. 2002. Indexing spatio-temporal data warehouses. In *Proceedings of the International Conference on Data Engineering* (*ICDE*) (Feb.), 166–175.

PIATETSKY-SHAPIRO, G. AND CONNELL, C. 1984. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the ACM SIGMOD Conference* (June), 256–276.

SALZBERG, B. AND TSOTRAS, V. 1999. A comparison of access methods for temporal data. *ACM Comput. Surv. 31*, 2, 158–221.

SOO, M., SNODGRASS, T., AND JENSEN, C. 1994. Efficient evaluation of the valid-time natural join. In *Proceedings of the International Conference on Data Engineering* (*ICDE*) (Feb.), 282–292.

TAO, Y. AND PAPADIAS, D. 2001a. The MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proceedings of the VLDB Conference* (Sept.), 431–440.

TAO, Y. AND PAPADIAS, D. 2001b. Efficient historical R-trees. In *Proceedings of the Scientific and Statistical Database Management* (*SSDBM*) (July), 223–232.

TAO, Y. AND PAPADIAS, D., ZHANG, J. 2002. Aggregate processing of planar points. In *Proceedings of the Extending Database Technology Conference* (*EDBT*) (March), 179–196.

THEODORIDIS, Y. AND SELLIS, T. 1996. A model for the prediction of R-tree performance. In *Proceedings of the ACM Symposium on Principles of Database Systems* (*PODS*) (June), 161–171.

THEODORIDIS, Y., STEFANAKIS, E., AND SELLIS, T. 2000. Efficient cost models for spatial queries using R-trees. *IEEE Trans. Knowl. Data Eng. 12*, 1, 19–32.

TZOURAMANIS, T., MANOLOPOULOS, Y., AND LORENTZOS, N. 1999. Overlapping B+-trees: An implementation of a transaction time access method. *Data Know. Eng.* 29, 381–404.

TZOURAMANIS, T., VASSILAKOPOULOS, M., AND MANOLOPOULOS, Y. 2000a. Overlapping linear quadtrees and spatio-temporal query processing. *Comput. J. 43*, 4, 325–343.

TZOURAMANIS, T., VASSILAKOPOULOS, M., AND MANOLOPOULOS, Y. 2000b. Multiversion linear quadtree for spatio-temporal data. In *Proceedings of the Database Systems for Advanced Applications Conference* (*DASFAA*) (Sept.), 279–292.

VARMAN, P. AND VERMA, R. 1997. An efficient multiversion access structure. *IEEE Trans. Knowl. Data Eng. 9*, 3, 391–409.

WEB. Http://dias.cti.gr/~ytheod/research/datasets/spatial.html.

XU, X., HAN, J., AND LU, W. 1990. RT-tree: An improved R-tree index structure for spatiotemporal data. In *Proceedings of the International Symposium on Spatial Data Handling Conference* (*SDH*) (July), 1040–1049.

YANG, J. AND WIDOM, J. 2001. Incremental computation and maintenance of temporal aggregates. In *Proceedings of the International Conference on Data Engineering* (*ICDE*) (April), 51–60.

YAO, S. 1978. Random 2-3 trees. *Acta Inf. 2*, 9, 159–179.

ZHANG, D., MARKOWETZ, A., TSOTRAS, V., GUNOPULOS, D., AND SEEGER, B. 2001. Efficient computation of temporal aggregates with range predicates. In *Proceedings of the ACM Symposium on Principles of Database Systems* (*PODS*) (May), 237–245.

ZHANG, D., TSOTRAS, V., AND SEEGER, B. 2002. Efficient temporal join processing using indices. In *Proceedings of the International Conference on Data Engineering* (Feb.), 103–113.

Received December 2001; revised May 2002; accepted July 2002