Cost Models for Overlapping and Multi-Version B-trees

Yufei Tao, Dimitris Papadias, Jun Zhang
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{taoyf, dimitris, zhangjun}@cs.ust.hk

Abstract

Overlapping and multi-version 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 multi-version techniques in several traditional (e.g., banking) and emerging (e.g., spatio-temporal) applications. In this paper, we propose frameworks for reducing performance analysis of overlapping and multi-version structures to that of the corresponding ephemeral structures, thus simplifying the problem significantly. The frameworks lead to accurate cost models that predict the sizes of the trees, the node accesses and query selectivity. Although we focus on B-tree-based structures, the proposed models can be employed with a variety of indexes.

1. Introduction

Versioning are those objects whose attributes change with time. Supporting such objects efficiently is crucial for a large number of applications. As an example, consider a banking 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 inquire about any time in history. Versioning databases constitute the core of many temporal, spatio-temporal, decision-making, and on-line 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 1.1 shows an example for the banking system. The vertical axis refers to account balances (i.e., the features), while the horizontal axis corresponds to 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 semi-closed 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 (e.g., a_2) is *alive* during its valid period (e.g. $[t_1, t_2)$), and *dead* outside it.

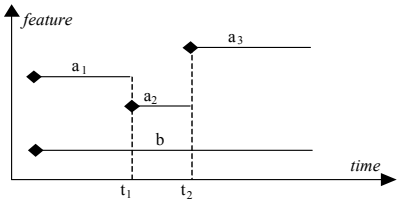


Figure 1.1: Representation of versioning objects

An important type of processing in versioning databases involves *range-interval queries* (*interval queries* for short), which consist of two predicates: (i) the time interval of interest and (ii) 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. The records retrieved by an interval query must be alive during the time interval and have their features in the feature range, e.g., "*find the accounts with balances greater than $1000 during March 2001*". When the time predicate involves only a single timestamp, the query is called a *range-timestamp query*.

Access methods for versioning objects have been extensively studied in temporal and spatio-temporal databases. Most existing methods are based on *multiple logical-tree structures* (MLTS). An MLTS maintains several logical trees, each of which is an *ephemeral structure* suitable for indexing objects at a single timestamp. To avoid excessive space, consecutive logical trees may share common branches so that these branches are stored only once. The *overlapping* and *multi-version* techniques are two popular frameworks for converting ephemeral structures into corresponding MLTS.

Little work has been carried out on analytical models for MLTS. Existing analysis merely discusses the asymptotic optimality of overlapping and multi-version B-trees with respect to timestamp queries. This, however, is insufficient in practice due to 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 cannot be used for other MLTS.

In this paper, we provide analytical *models that, in addition to B-trees, can be employed for any MLTS provided that there exists an analytical model for the corresponding ephemeral structure*. For instance, in order to analyze the performance of an overlapping (or multi-version) structure based on R-trees, we only need to incorporate the corresponding R-tree models into our framework to obtain the cost models for the MLTS. The proposed models can accurately predict: (i) the node accesses in performing interval and timestamp queries; (ii) tree sizes; (iii) query selectivity. Furthermore, the formulae are based only on the properties of the raw data and the underlying file system; hence, they do not require knowledge about the structure of the trees.

The rest of the paper is organized as follows: Section 2 surveys overlapping and multi-version structures and describes in detail the two frameworks using B-trees as the ephemeral structures. Section 3 presents the cost models for B-tree-based methods. Section 4 contains an extensive experimental evaluation to prove the efficiency of the proposed models. Section 5 concludes the paper with future directions.

2. Overlapping and Multi-version Methods

The overlapping technique was first introduced in [BH85] to produce a time and space efficient approach to file sharing. The idea was applied to B-trees in [BKK+90] and R-trees [NS98]. The resulting structures were called overlapping B-trees (OVB-trees) and historical R-trees (HR-trees), respectively. Tzouramanis et al. [TML99] extended OVB-trees by integrating pointers among leaf pages, which improve the so-called *key-history* queries in temporal databases. Recently, the technique was also applied on Linear Quadtrees [TVM00a]. In a survey paper [ST97], Salzberg and Tsotras compared asymptotic performance of overlapping methods with other temporal access methods in terms of timestamp query performance, update costs, and structure sizes. Interval query performance was not discussed.

The multi-version framework was initiated by [BGO+96], which proposed the Multi-version B-tree (MVB-tree) and proved its optimal asymptotic performance. Varman and Verma [VV97] presented a variation of MVB-trees, which reduces the size requirements by some constant factor. The multi-version framework has produced many efficient access methods in various scenarios. Methods based on R-trees include BTR-trees to index bitemporal databases [KTF98], and PPR-trees [KGT] and MVR-trees [TP01] for spatio-temporal databases. Multi-version linear Quadtrees were proposed for image processing in [TVM00b]. Furthermore, [JSL+00] and [ZMT+01] employed the idea in branched temporal databases and temporal aggregation respectively to obtain BT-trees and multi-version SB-trees. To the best of our knowledge there does not exist any work that estimates the structure sizes and performance of multi-version methods in terms of node accesses when performing interval queries.

In the rest of the section, we describe the overlapping and multi-version 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 underlying records do not change. Insertion and deletion are carried out in a way similar to 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 2.1 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, while e_1 should be inserted. As shown in the figure, 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.

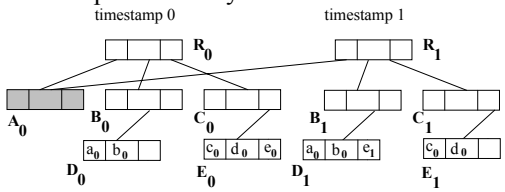


Figure 2.1: An OVB-tree example

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*" proposed in [TP01].

2.2 Multi-version B-trees

In multi-version structures, each entry has the form <K, t_{st}, t_{ed}, *pointer*> where t_{st} (the *insertion time*) denotes the time that the record was inserted in the database, and t_{ed} (the *deletion time*) denotes the time that it was deleted[1]. For leaf entries, K denotes the features of an object (e.g., the balances of accounts). For an intermediate entry, K 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 [BGO+96], K equals the minimum value of features in the subtree (the bounding range can be derived by considering the value of K in the next entry). 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 "*" (denoting 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 2.2 illustrates an example of an MVB-tree.

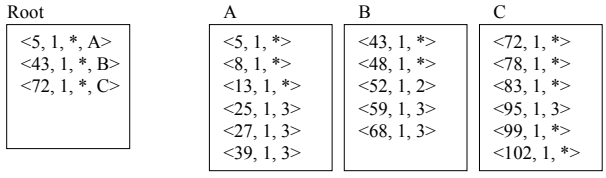


Figure 2.2: A MVB-tree example

For each timestamp t and each node except the roots, it is required that either none, or at least $b \cdot P_{version}$ entries are alive at t, where $P_{version}$ is a tree parameter and b the node capacity (for the following examples $P_{version}$=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). The insertion of <28,4,*> into node A at timestamp 4 (in the tree of Figure 2.2) will cause node A to overflow. A new node D is created to store the live entries of A, and A "dies" meaning that it will not be modified any more in the future. A new entry <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.

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 will 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_{svu} and P_{svo} are tree parameters). 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.

Strong version underflow is similar to weak version underflow, the only difference being that the former happens after a version split, while 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. In [VV97], 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 query starts by retrieving the corresponding roots whose jurisdiction intervals intersect the queried interval. Then search is guided by K, t_{st}, and t_{ed} till the leaves. As with 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 [BS96].

As shown in [BGO+96], for n versions produced by N objects, a multi-version structure consumes O(n) space and answers timestamp queries in O($\log N + k/b$), where b is the node capacity and k is the number of records retrieved. In spite of the asymptotic optimality, multi-version structures do not answer timestamp queries as efficiently as overlapping structures, which, however, consume O($n\log N$) space [ST97].

3. Performance Analysis of MLTS

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

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

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

$$a_i = \frac{k}{N}$$

In order to facilitate analysis, it is common to make some assumptions, such as independence of predicates, uniformity etc. In case of R-trees, for instance, cost models usually assume that objects are uniformly distributed in the spatial universe [PSTW93, PS96]. The derived formulae can then be combined with statistical information, such as histograms, to deal with real data [TSS00]. Following the same approach, we will make similar assumptions for MLTS and will later discuss how they can be extended for general cases.

Without loss of generality, we consider that features of objects distribute in a unit universe $[0, 1]^d$, where d is the dimensionality of the feature universe (for B-trees, d=1). Formally, we define the problem of cost model analysis for MLTS as follows: at the first timestamp (timestamp 1), the features of N objects distribute in the unit range [0, 1) by a certain distribution *DIST*. At each of the subsequent timestamps i (i = 2, 3, ..., T), a% of the N objects issue feature changes, where T corresponds to the total number of timestamps recorded so far (i.e., the data agility remains constant). The changes of the objects are in such a way that (i) the distribution of objects' features does not change (i.e., conforms to *DIST*) at each timestamp; (ii) each object has the same probability to produce changes. We index the dataset with an overlapping or multi-version structure, and the goal is to predict the expected number of node accesses in answering a query. Note that we fix the cardinality of the dataset simply to prevent excessively complex results. Our approach can be extended easily to support arbitrary changes of the cardinality (i.e., the number of insertions is different from that of deletions at one timestamp).

3.1 A Unified Cost Model

An interval query q can be represented as $q(q_k, q_t)$ to indicate its feature range predicate q_k and temporal interval predicate q_t. 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 features of entries in node s, while the interval projection s_t corresponds to 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, since 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. Recall that objects change in such a way that their feature distribution remains the same at any timestamp. Therefore, 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 involves a deletion followed by an insertion, and that every object has the same probability to issue changes leads to two important observations: (i) live nodes at the same tree level receive approximately the same number of insertions (deletions) at each timestamp; (ii) the number of live entries in a node remains constant throughout its lifespan. As a result, after the first logical tree is constructed, node duplication and version split become the major types of structural changes for OVB-trees and MVB-trees respectively. This is supported by our experiments: starting from the second timestamp, the number of key splits (weak version underflows and strong version overflows/underflows) is significantly smaller than the number of node duplications (version splits) for OVB- (MVB-) trees. Therefore, in the cost model 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 *evolution rate* as in Def 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 will 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.

Def 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:

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

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

$$prob(s, q) = prob_{feature} \cdot prob_{tm} \quad (3.1\text{-}1)$$

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

$$prob_{feature} = \prod_{i=1}^{d} (s_k^i + q_k^i) \quad (3.1\text{-}2)$$

where s_k^i and q_k^i are the extents along the ith dimension for s_k and q_k respectively [TSS00].

For the special case, when the dimensionality is 1, the estimation for $prob_{feature}$ becomes:

$$prob_{feature} = s_k + q_k \quad (3.1\text{-}3)$$

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:

$$prob_{tm} = \frac{P_i}{K_i} \quad (3.1\text{-}4)$$

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

$$K_i = M_i + E_i M_i (T-1) \quad (3.1\text{-}5)$$

where T denotes the total number of timestamps in history. The reasoning behind equation (3.1-5) 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 one dies, so the lifespans of these two nodes are disjoint and continuous. Figure 3.1 shows a query whose temporal interval intersects the lifespans of 4 nodes, where a_2 and a_3 were created when a_1 and a_2 died respectively.

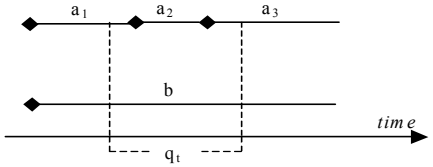


Figure 3.1: Time evolution of nodes

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 (let time i) of q_t. Then, at each of the subsequent timestamp $i + j$ ($1 \le j \le |q_t| - 1$), $E_i \cdot M_i$ nodes are created. Hence, we have the following estimation for P_i:

$$P_i = M_i + E_i M_i (q_t - 1) \quad (3.1\text{-}6)$$

Combining equations (3.1-4), (3.1-5), (3.1-6), we have:

$$prob_{tm} = \frac{1 + E_i (q_t - 1)}{1 + E_i (T-1)} \quad (3.1\text{-}7)$$

Extending equations (3.1-1) and (3.1-3), we derive:

$$prob(s,q) = (s_k + q_k) \frac{1 + E_i (q_t - 1)}{1 + E_i (T-1)} \quad (3.1\text{-}8)$$

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 $NA(q)$ in answering query q is given by the following 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 equation (3.1-9), where h denotes the height of a logical tree and K_i denotes the total number nodes at level i. Again, we emphasize that this equation applies to general MLTS as well, except that equation (3.1-2) should be used for $prob_{feature}$.

$$NA(q) = \sum_{i=0}^{h-1} \left[K_i \cdot prob(s_i, q) \right] \quad (3.1\text{-}9)$$

where $prob(s_i, q)$ is given by equation (3.1-8).

In this work, we assume each node in the trees occupies a single disk page; hence equation (3.1-9) 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 dataset indexed 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 (3.1-9) 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 (3.2-1), where f is the fanout of the tree. The commonly adopted value for f is $ln2 \cdot b \approx 0.69b$ [Yao77], where b is the node capacity. Note that since the node capacity is decided by the page size of the underlying file system, the value of f is independent of the dataset indexed. Further, since each logical tree indexes the same number of objects, the height of each tree is expected to be the same.

$$h = 1 + \lceil \log_f (N/b) \rceil \quad (3.2\text{-}1)$$

- **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 dataset 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:

$$E_0 = 1-(1-\frac{f}{N})^{2aN} \quad (3.2\text{-}2)$$

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 (3.2-2), we obtain:

$$E_i = 1-(1-\frac{f^{i+1}}{N})^{2aN} \quad (0 \le i \le h-1) \quad (3.2\text{-}3)$$

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:

$$K_i = \frac{N}{f^{i+1}} + E_i\frac{N}{f^{i+1}}(T-1) \quad (3.2\text{-}4)$$

Note that a corollary of equation (3.2-4) is that we can estimate the size of an OVB-tree as follows:

$$Size(OVB) = \sum_{i=0}^{h-1} K_i = \sum_{i=0}^{h-1}\left[\frac{N}{f^{i+1}} + E_i\frac{N}{f^{i+1}}(T-1)\right] \quad (3.2\text{-}5)$$

- **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 (existing analysis usually focuses on uniform distribution), 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:

$$s_{ik} = \frac{f^{i+1}}{N} \quad (0 \le i \le h-1) \quad (3.2\text{-}6)$$

So far we have rewritten all the components of equation (3.1-9) as functions of f, N, a, D, and T. The final number of node accesses during query processing, is presented in Equation (3.2-7), where E_i is given by equation (3.2-3). Note that when E_i is 0, the above equation degenerates into a cost model for conventional B+-trees.

$$NA(q) = \sum_{i=0}^{h-1}\left[\frac{N}{f^{i+1}} + E_i\frac{N}{f^{i+1}}(T-1)\right]\left(\frac{f^{i+1}}{N}+q_k\right)\frac{1+E_i(q_t-1)}{1+E_i(T-1)} \quad (3.2\text{-}7)$$

3.3 A Cost Model for MVB-trees

In the sequel, we carry out a similar analysis for MVB-trees based on equation (3.1-9).

- **Estimating h**

Let f_l be the average number of live entries at a single timestamp in node s. Note that f_l is different from f, which equals the total number of entries in s. Thus, the height of a logical tree is given by equation (3.3-1):

$$h = 1+\lceil \log_{fl}(N/b) \rceil \quad (3.3\text{-}1)$$

Meanwhile, let M_i denote the average number of level i nodes that are alive at a single timestamp in a logical tree:

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

The estimation for f_l 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. Thus, f_l should approximate the fanout of a B-tree whose node capacity is $b\cdot P_{svo}$. Hence, we estimate f_l as $ln2\cdot b\cdot P_{svo}$, which is shown to be accurate through experiments.

- **Estimating E_i**

We first present the estimation for E_0. A node s contains f_l entries when it is created from a version split; thus, s will receive $(b-f_l)$ 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 aN/M_0 insertions. As a result, s will generate a version split after $(b-f_l)M_0/(aN)$ timestamps. Since $M_0 = N/f_l$, the number of timestamps s_{0t} that a leaf level node remains alive before it is version split, can be estimated as follows:

$$s_{0t} = \frac{(b-f_l)M_0}{aN} = \frac{b-f_l}{af_l}$$

Let V_i be the total number 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:

$$V_0 = M_0 \frac{T-1}{s_{0t}} = \frac{aN(T-1)}{b-f_l}$$

$$v_0 = \frac{V_0}{(T-1)} = \frac{aN}{b-f_l}$$

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:

$$E_0 = \frac{v_0}{M_0} = \frac{af_l}{b-f_l}$$

Whenever a leaf node generates a version split, an entry will be inserted into its parent node at level 1. Hence, 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 $(b-f_l)M_1 / v_0$ timestamps after its creation. Therefore, the lifespan of the node, s_{1t} is:

$$s_{1t} = \frac{(b-f_l)M_1}{v_0} = \frac{(b-f_l)^2}{af_l^2}$$

The estimation for V_1 is as follows:

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

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

$$s_{it} = \frac{(b-f_l)^{i+1}}{af_l^{i+1}} \qquad (0 \le i \le h-1)$$

$$V_i = \frac{aN(T-1)}{(b-f_l)^{i+1}} \qquad (0 \le i \le h-1) \qquad (3.3\text{-}2)$$

Hence, we have:

$$v_i = \frac{V_i}{(T-1)} = \frac{aN}{(b-f_l)^{i+1}}$$

$$E_i = \frac{v_i}{M_i} = \frac{af_l^{i+1}}{(b-f_l)^{i+1}} \qquad (3.3\text{-}3)$$

- **Estimating K_i**

Given that the total number of version splits at level i is provided by equation (3.3-2), the total number K_i of nodes created through history is:

$$K_i = M_i + V_i = \frac{N}{f_l^{i+1}} + \frac{aN(T-1)}{(b-f_l)^{i+1}} \quad (0 \le i \le h-1) \quad (3.3\text{-}4)$$

As a corollary of equation (3.3-4), the size of an MVB-tree can be estimated as:

$$Size(MVB) = \sum_{i=0}^{h-1} K_i = \sum_{i=0}^{h-1} \left[\frac{N}{f_l^{i+1}} + \frac{aN(T-1)}{(b-f_l)^{i+1}} \right] \qquad (3.3\text{-}5)$$

- **Estimating s_{ik}**

As mentioned above, a node contains f_l live entries at the same timestamp. Therefore, replacing f in equation (3.2-6) for OVB-trees with f_l, we obtain the following equation for the average key range of nodes at level i:

$$s_{ik} = \frac{f_l^{i+1}}{N} \qquad (0 \le i \le h-1) \qquad (3.3\text{-}6)$$

Equation (3.3-7) presents the final model, which predicts the node disk accesses for range-interval queries based on the properties of the dataset indexed and the underlying file system (E_i is estimated by equation (3.3-3)).

$$NA(q) = \sum_{i=0}^{h-1} \left[\frac{N}{f_l^{i+1}} + \frac{aN(T-1)}{(b-f_l)^{i+1}} \right] \left(\frac{f_l^{i+1}}{N} + q_k \right) \frac{1 + E_i(q_t - 1)}{1 + E_i(T-1)} \qquad (3.3\text{-}7)$$

3.4 Estimation for Query Selectivity

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 (3.1-8), except that (i) the *evolution rate* of the objects now corresponds to the agility a of the dataset; (ii) i_k is set to 0 because the feature of each entry indexed in an OVB- or MVB-tree contains only a single value. Hence, we have:

$$sel(q) = prob(i,q) = q_k \frac{1 + a(q_t - 1)}{1 + a(T-1)} \qquad (3.4\text{-}1)$$

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

$$\begin{aligned} NUM(q) &= prob(i,q)\left[N + aN(T-1)\right] \\ &= N \cdot q_k \left[1 + a(q_t - 1)\right] \end{aligned} \qquad (3.4\text{-}2)$$

3.5 Predicting the Behaviour of MLTS

The proposed models can answer two important questions: (i) when it is worth using a MLTS instead of ephemeral structures (e.g., an independent B-tree for each timestamp); (ii) which MLTS is preferable in terms of structure size and query performance considerations. Regarding the first question, notice that when the agility exceeds a certain threshold (which we call *degradation agility*), all the live nodes will be duplicated (version split) in an overlapping (multi-version) structure, at each timestamp; i.e., 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 (3.2-3) and (3.3-3) for OVB- and MVB-trees respectively. Solving these equations, we obtain the degradation agilities for OVB- and MVB-trees as follows:

$$deg\text{-}agility(OVB) \approx \frac{-1}{N \log(1 - f/N)} \qquad (3.5\text{-}1)$$

$$deg\text{-}agility(MVB) \approx \frac{b-f_l}{f_l} = \frac{1 - \ln 2 \cdot P_{svo}}{\ln 2 \cdot P_{svo}} \qquad (3.5\text{-}2)$$

For OVB-trees the estimated degradation agility is very low (around 5% for our experimental settings), which severely limits their applicability. In order to intuitively

explain this phenomenon, consider a typical situation where the average fanout of OVB-trees is f = 83.82 (this number is used in our experiments). Even if one (out of f) object in a node issues a change, the node must be copied (which leads to replication of all f 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/f objects issue updates at a timestamp, an OVB-tree may degenerate to independent B-trees.

On the other hand, although the estimated degradation agility of MVB-trees is more than an order of magnitude higher (81% 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 multi-version 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 *multi-tree point* (*MTP*, for short), it consumes more space than the equivalent ephemeral B-trees. The MTP can be predicted by equation (3.5-3), which compares the size of an MVB-tree to that of independent B-trees.

$$Size(MVB) = T \cdot \sum_{i=0}^{\lceil \log_f N \rceil - 1} \frac{N}{f^{i+1}} \qquad (3.5\text{-}3)$$

where $Size(MVB)$ is given in equation (3.3-5), and f is equal to the fanout of a B- (or OVB-) tree. For our settings the estimated MTP is 33%, meaning that above this agility it is preferable to build ephemeral B-trees.

As mentioned, the OVB-tree is applicable only for very low agilities. Even below the degradation agility, its size is expected to be much larger than that of the corresponding MVB-trees due to extensive replication. The only reason for using OVB-trees (or overlapping data structures, in general) is when the workload consists mainly of timestamp queries. OVB-trees are more efficient than MVB-trees for timestamp queries because of their higher fanout. On the other hand, MVB-trees are more efficient for interval queries, and the performance gain increases with the query length (for the same q_k). This can be explained by observing equations (3.2-7) and (3.3-7). 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, by (3.2-7) and (3.3-7), we have:

$$\frac{NA_i(OVB)}{NA_i(MVB)} = \frac{1 + E_{Oi}(q_t - 1)}{1 + E_{Mi}(q_t - 1)} \cdot C$$

where C is a constant, 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 increases with q_t. These observations are experimentally evaluated in the next section.

4. Experimental Evaluation

In this section, we demonstrate the efficiency of the proposed models through experimental results. Datasets were created as follows. At the first timestamp, the 1D features (each feature is a single value) of 20,000 objects are uniformly generated in the universe [0,1). Then, at each of the following 200 timestamps, a% of the objects are selected to produce feature changes so that the distribution of the keys is still uniform at each timestamp. The agility varies for different datasets; we refer to a dataset with agility a as $DS_{a\%}$.

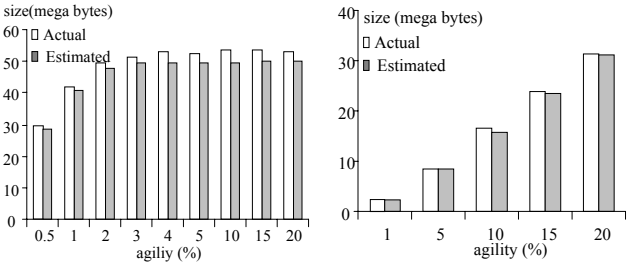
Query performance is measured by the average node accesses in answering *workloads*, each consisting of 500 queries. All the queries in a workload involve a feature range of the same length q_k and an interval range with the same number of timestamps q_t. The left end points of the feature and time ranges 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 $WRKLD_{qk,qt}$ to indicate its parameters.

We experimented with a wide range of parameters. To demonstrate the effects of several factors on performance, the settings for the following experiments are chosen as follows: (i) The values for a are 0.5%, 1%, 2%, ..., 5%, 10%, 15%, and 20%, resulting in datasets with 40,000 to 800,000 records; (ii) values for q_k range from 0.1 to 0.5 (i.e., 10% to 50% of the feature space); (iii) values for q_t range from 1 to 20 timestamps (0.5% to 10% of the entire history).

OVB-trees and MVB-trees were implemented as described in [ST97] and [BGO+96] respectively. The parameters for MVB-trees are as follows: $P_{version}$ = 0.2, P_{svo} = 0.8, and P_{svu} = 0.4. The page size is set to 1,024 bytes in all cases. With this size, the node capacities of OVB- and MVB-trees are 122 and 62 entries. Hence, for OVB-trees, f = 83.82, while, for MVB-trees, f_l = 34.3. These values are used for the corresponding cost models.

We first evaluate equations (3.2-5) and (3.3-5) on the sizes of the MLTS. Figures 4.1 (a) and (b) show the sizes of OVB- and MVB-trees as a function of agility. OVB-trees initially grow very fast, and their size stabilizes after the degradation agility (5%) where they degenerate into independent trees. On the other hand, MVB-trees are much more space-efficient for normal agilities (up to 20%). In both cases the estimation is accurate.

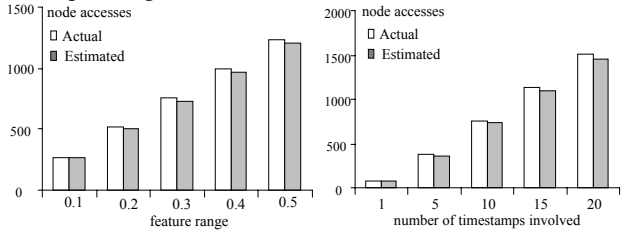
In order to verify the estimated degradation agility for MVB-trees (81%), we increased the agility 5% at a time while checking if there is noticeable increase in the tree sizes. We found that the sizes of MVB-trees stabilize at around 130 megabytes when the agility becomes 80%. Above 35% agility, MVB-trees consume more space than independent B-trees (i.e., around 50 megabytes in Figure 5.1), which is consistent with the estimated value of MTP (33%) obtained by equation (3.5-3).



(a) Sizes of OVB-trees (b) Sizes of MVB-trees

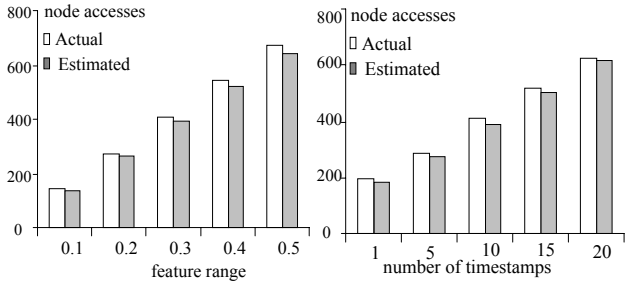
Figure 4.1: Sizes of MLTS as a function of agility

To evaluate the estimation of node accesses (NA), provided by equations (3.2-7) and (3.3-7), with respect to different query parameters, we fix the agility of the datasets indexed by OVB- and MVB-trees to 2% and 10% respectively. Note that the selected agility for the dataset indexed by the OVB-tree is relatively lower to prevent its degradation (in which case the cost estimation is trivial). Then, we performed the following two sets of queries: (i) for workloads in the first set we fix q_k to 0.3 (30% of the feature space), and vary q_t from 1 to 20 timestamps; (ii) for workloads in the second set we fix q_t to 10, and vary q_k from 0.1 to 0.5. Figures 4.2 (a) and (b) demonstrate the results for OVB-trees with respect to the two sets of workloads respectively. The node accesses measured are averaged over the total number of queries for each workload. Figures 4.3 (a) and (b) show the corresponding results for MVB-trees.



a) NA for $WRKLD_{0.1\sim0.5,\ 10}$ (b) NA for $WRKLD_{0.3,\ 1\sim20}$

Figure 4.2: Query cost for OVB-trees (agility = 2%)

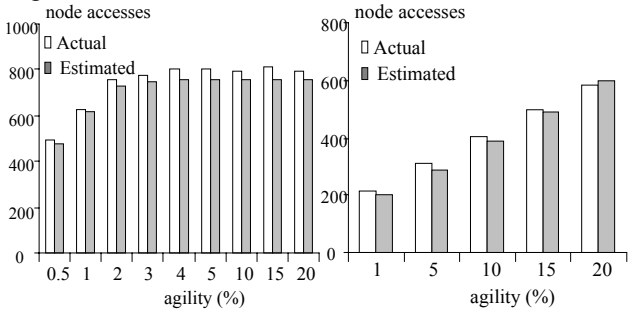


(a) NA for $WRKLD_{0.1\sim0.5,\ 10}$ (b) NA for $WRKLD_{0.3,\ 1\sim20}$

Figure 4.3: Query cost for MVB-trees (agility = 10%)

Notice that the query cost increases linearly with both the number of timestamps involved and the lengths of the feature ranges, as predicted by equations (3.2-7) and (3.3-7). As expected, OVB-trees are more efficient than MVB-trees only for timestamp queries. The superiority of MVB-trees increases with the query length.

The next set of experiments evaluates the cost models when the dataset agility varies. Specifically, we measured the query performance of $WRKLD_{0.3,10}$ for the corresponding OVB- and MVB-trees using datasets with agilities from 1% to 20%. According to Figures 4.4 (a) and (b) the query costs follow similar trends with the tree sizes and MVB-trees perform better than OVB-trees for the agilities demonstrated. Notice, however, that the cost of MVB-trees increases linearly with the agility, while that of OVB-trees is stable after their degradation, since each query accesses the same number of independent B-tress (i.e., 10), independently of the agility. Above the MTP, the query cost of MVB-trees exceeds that of degraded OVB-trees.

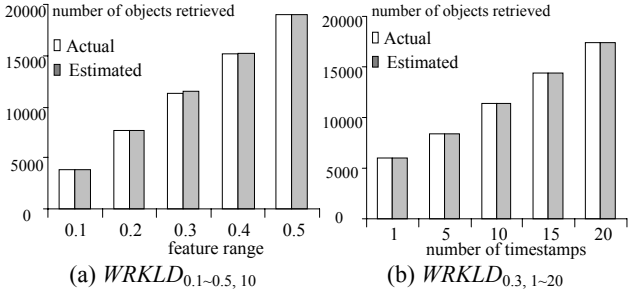


(a) OVB-trees (b) MVB-trees

Figure 4.4: Query vs. agilities (for $WRKLD_{0.3,\ 10}$)

For most of the cases above, the cost models slightly underestimate the actual results. This is expected because the analysis focuses on the major structural changes. Consider the size of MVB-trees as an example. If a node incurs a strong version overflow after a version split, two new nodes are generated, instead of one as assumed by the cost models. Similarly, when an OVB-node that has already been duplicated at the current timestamp incurs an overflow, it will be key split rather than duplicated. However, as shown, structural changes other than those considered in the analysis are very infrequent; ignoring them does not introduce significant error.

Finally, we evaluate the efficiency of equation (3.4-2) towards the selectivity estimation. In Figures 4.5 (a) and (b), the dataset agility is fixed to 10% and we vary q_k and q_t. The y-axis of all the figures corresponds to the number of interval records retrieved in the query. The experimental and estimated results are almost identical.



(a) $WRKLD_{0.1\sim0.5,\ 10}$ (b) $WRKLD_{0.3,\ 1\sim20}$

Figure 4.5: Selectivity estimation

5. Conclusion

Our approach reduces the cost analysis of MLTS to that of the corresponding ephemeral structures, which means that our framework is applicable to a variety of different access methods. Extensive experimentation proves the accuracy of the models for a wide range of conditions. To the best of our knowledge, this is the first work that attempts to provide systematic analysis for these types of structures. Given, the ever increasing availability and importance of historical data in numerous applications, accurate 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 insights into the behavior of overlapping and multi-version structures. Our analysis predicts, and the experimentation verifies that: (i) OVB-trees are meaningful only very small agilities (about 5% for our settings), since they quickly degenerate into multiple trees. Even below their degradation agility, they perform better than MVB-trees only for timestamp (or very short interval) queries. (ii) MVB-trees are best for agilities up to about 30%, since they consume less space than OVB-trees (or ephemeral B-trees) and perform better for mixed workloads. (iii) For agilities above the MTP, the best choice is to build an independent structure for each timestamp. The proposed models can accurately estimate the above agility thresholds depending on the system characteristics

Future work can deal with skewed data, possibly with the aid of statistics. As an example, histograms have been widely employed for traditional and spatial data. In our case, the application of histograms is not straightforward, since in addition to the total number of records and the distribution of the key values per timestamp, we may need to keep additional information regarding how the values change. For example, 1000 insertions plus 500 deletions at one timestamp have very different impact from 500 insertions without deletion on the tree structures, though both cases lead to an increment of 500 in object cardinality. Furthermore, although we only focused on range queries, this work can serve as the basis for performance analysis of other more complex queries, such as temporal joins.

Acknowledgments

This work was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

References

[BGO+96] Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal*, Vol. 5(4), pp. 264-275, 1996.

[BH85] Burton, F.W., Huntbach, M.M. Multiple Generation Text Files Using Overlapping Tree. *The Computer Journal*, Vol. 28, No. 4, pp. 414-416, 1985.

[BKK+90] Burton, F.W., Kollias, J.G., Kollias, V.G., Matsakis, D.G. Implementation of Overlapping B-trees for Time and Space Efficient Representation of Collection of Similar Files. *The Computer Journal*, Vol. 33(3), pp. 279-280, 1990.

[BS96] Bercken, J.v.d., Seeger, B. Query Processing Techniques for Multiversion Access Methods. *VLDB*, 1996.

[JSL+00] Jiang, L., Salzberg, B., Lomet, D., Barrena, M. The BT-Tree: A Branched and Temporal Access Method. *VLDB*, 2000.

[KF93] Kamel, I., Faloutsos, C. On Packing R-trees. *CIKM*, 1993.

[KGT] Kollios, G., Gunopulos, D., Tsotras, V., Delis, A., Hadjieleftheriou, M. Indexing Animated Objects Using Spatiotemporal Access Methods. *To appear in IEEE TKDE*.

[KTF98] Kumar, A., Tsotras, V.J., Faloutsos, C. Designing Access Methods for Bitemporal Databases. *IEEE TKDE*, Vol. 10, No. 1, pp. 1-20, 1998.

[NS98] Nascimento, M., Silva, J. Towards Historical R-trees. *ACM SAC*, 1998.

[PS96] Pagel, B., Six, H. Are Window Queries Representative for Arbitrary Range Queries? *PODS*, 1996.

[PSTW93] Pagel, B.-W., Six, H., Toben, H., Widmayer, P. Towards an Analysis of Range Query Performance. *PODS*, 1993.

[ST97] Salzberg, B., Tsotras, V. A Comparison of Access Methods for Temporal Data. *ACM Computing Surveys*, 31(2): 158-221, 1997.

[TML99] Tzouramanis, T., Manolopoulos, Y., Lorentzos, Overlapping B+-trees: An Implementation of a Transaction Time Access Method. *Data Knowledge and Engineering*, Vol. 29, pp. 381-404, 1999.

[TP01] Tao, Y., Papadias, D. The MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. *VLDB*, 2001.

[TSS00] Theodoridis, Y., Stefanakis, E., Sellis, T. Efficient Cost Models for Spatial Queries Using R-Trees. *IEEE TKDE*, Vol. 12, (1), pp. 19-32, 2000.

[TVM00a] Tzouramanis, T., Vassilakopoulos, M., Manolopoulos, Y. Overlapping Linear Quadtrees and Spatio-Temporal Query Processing. *The Computer Journal*. Vol. 43(4), pp. 325-343, 2000.

[TVM00b] Tzouramanis, T., Vassilakopoulos, M., Manolopoulos, Y. Multiversion Linear Quadtree for Spatio-Temporal Data. *DASFAA*, 2000.

[VV97] Varman, P., Verma, R. An Efficient Multiversion Access Structure. *IEEE TKDE*, Vol. 9(3), pp. 391-409, 1997.

[Yao77] Yao, S. Approximating Block Accesses in Database Organizations. *Communications of the ACM*, Vol. 20, No. 4, pp. 260-261, 1977.

[ZMT+01] Zhang, D., Markowetz, A., Tsotras, V., Gunopulos, D., Seeger, B. Efficient Computation of Temporal Aggregates with Range Predicates. *PODS*, 2001.