ELSEVIER Data & Knowledge Engineering 29 (1999) 381–404

DATA & KNOWLEDGE ENGINEERING

Overlapping B^+-trees: An implementation of a transaction time access method

Theodoros Tzouramanis[1], Yannis Manolopoulos[1,*], Nikos Lorentzos[2]

[1]*Department of Informatics, Aristotle University, 54006 Thessaloniki, Greece*
[2]*Laboratory of Informatics, Agricultural University, 11855 Athens, Greece*

Received 6 February 1998; revised 21 August 1998; accepted 29 September 1998

Abstract

A new variation of Overlapping B^+-trees is presented, which provides efficient indexing of transaction time and keys in a two dimensional key-time space. Modification operations (i.e. insertions, deletions and updates) are allowed at the current version, whereas queries are allowed to any temporal version, i.e. either in the current or in past versions. Using this structure, snapshot and range-timeslice queries can be answered optimally. However, the fundamental objective of the proposed method is to deliver efficient performance in case of a general pure-key query (i.e. 'history of a key'). The trade-off is a small increase in time cost for version operations and storage requirements. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Temporal databases; Transaction time; Access methods; Indexing; Algorithms; Time and space performance

1. Introduction

Conventional databases usually store only one version of data, that one which is applicable at *only the present time*. Therefore, whenever a piece of data is not valid any longer, it is either deleted or updated, at the physical level. For ease of discussion, such databases are called *snapshots*. In recent years, however, there is an increasing interest in maintaining many time-varying data versions and in supporting queries against them or in performing trend analysis for decision making [11]. Systems supporting these requirements are called *temporal databases* (TDBs).

Two types of time are usually considered in TDBs, *valid* and *transaction time*. (A third type of time, *user-defined*, is treated as any other attribute of a conventional relation; thus it does not require some sort of special support [19].) According to [7] valid time is the time during which a fact is true in the real world. Transaction time is the time during which a piece of data is recorded in the relation. 'Transaction times are consistent with the serialization order of transactions and can be implemented'

* Corresponding author. E-mail: yannis@delab.csd.auth.gr

as a single value by 'using the commit times of transactions' [18]. In terms of data modeling, these two types of time usually comprise a *start time point* and an *end time point* or, equivalently, a *period* [*StartTime*, *EndTime*).[1] A TDB handling only valid time is called *valid* or *historical*; when it handles only transaction time it is called *transaction* or *rollback*; one handling both of these times is called *bi-temporal* [8].

To illustrate the difference between the above-mentioned types of databases, consider the following three facts:

- F_1. John is hired with a salary of 10 k,
- F_2. His salary increases to 15 k, and
- F_3. John retires.

Assume also that the facts F_1, F_2, F_3 become true (valid) in the real world at times V_{10}, V_{20} and V_{30} and that they are recorded in the database at times T_{15}, T_{25} and T_{35}, respectively. It is now shown how these facts are recorded in the various types of databases.

In the case of a snapshot database, the interest restricts to the recording of the data that is valid *only at the present time*. Hence, at time T_{15}, a tuple, (John, 10 k), is recorded in the snapshot database, in response of F_1. This tuple remains recorded until time T_{25}, when it is replaced by tuple (John, 15 k), in response of F_2. This last tuple remains recorded until time T_{35}, when it is deleted, in response of F_3.

In the case of a valid time database, the interest extends to the recording of both the pieces of data that are valid at times other than the present, along with their respective *validity times*. Hence, at time T_{15} (F_1), the tuple (John, 10 k, [V_{10}, *UC*)), is recorded in a *valid time relation*. This tuple indicates that John's salary became 10 k at time V_{10} and will remain the same *Until it is Changed* (*UC*). It remains recorded until time T_{25} (F_2), when it is replaced by two tuples (John, 10 k, [V_{10}, V_{20})) and (John, 15 k, [V_{20}, *UC*)). The former indicates that John's salary was 10 k during the period [V_{10}, V_{20}), whereas the latter indicates that John's salary became 15 k at time V_{20} and will remain so until it is also changed. Finally, at time T_{35} (F_3) this second tuple is replaced by (John, 15 k, [V_{20}, V_{35})). Note that the values for valid time, V_{10}, V_{20} and V_{30}, are supplied by the user.

In the case of a transaction time database, the interest concerns the recording of data along with its respective transaction time, i.e. the time during which this data remained physically in the database. Hence, at time T_{15} (F_1) the tuple (John, 10 k, [T_{15}, *now*)), is recorded in a *transaction time relation*. It indicates that the data (John, 10 k) was physically stored in the database at time T_{15} and it remains recorded until the *current* time. At time T_{25} (*F_2*), *this tuple is replaced by two tuples*, (*John*, 10 *k*, [T_{15}, T_{25})) and (John, 15 k, [T_{25}, *now*)). Now, the former indicates that the data (John, 10 k) was physically recorded at time T_{15} and at time T_{25} it was *logically* deleted. It is noted therefore that the original tuple is not deleted *physically* from the *transaction time relation*; instead, the end of its period is replaced by T_{25}, the time at which the user's update was executed. In a similar manner, at time T_{35} (F_3) the above second tuple is replaced by (John, 15 k, [T_{25}, T_{35})). Note that, as opposed to valid time databases, the values for the transaction time, T_{15}, T_{25} and T_{35} of such a database, are supplied by the DBMS. Specifically, whenever a user issues an insertion, deletion or update operation, the DBMS accesses the present time from the computer's clock and records it automatically along with the data that the user supplies.

Finally, in the case of a bi-temporal database, both the valid and transaction time are recorded along with the remainder data in a *bi-temporal relation*. As an example, at time T_{15}, a bi-temporal database

[1] A semi-closed time interval [x, y) includes all time instants t that satisfy $x \leq t < y$.

contains a tuple, (John, 10 k, $[V_{10}, UC)$, $[T_{15}, now)$). The maintenance of a bi-temporal database is more complicated than all other types of databases. The remainder modification operations of such a database are not given, since they are beyond the objectives of the present paper.

As opposed to work at the design of the logical level of a TDB (see for example [12,13,20], work at the design of the physical level is less extensive. In general, problems at this level are complex; i.e. special access methods and efficient query optimization techniques have to be proposed in order to achieve acceptable performance in real-life applications. A number of methods which have been proposed can be found in the surveys [16,18]. In work relevant to access methods, a query is classified [18] according to the notation: *Key/Valid/Transaction.* This notation specifies which entries are involved in the query and in what manner. Each entry can take one of the values: 'point', 'range', '*' or '−'. A 'point' for the *Key* entry means that the user has specified a single value to match the key attribute, while 'point' for the *Valid* or *Transaction* entry implies that a time instant value is specified for this entry. Similarly, 'range' indicates a specified range of values for the *Key* entry, or a period for the *Valid* or *Transaction* entry. A '*' means that any value is accepted in the entry, while '−' means that this entry is not applicable. Since transaction databases do not support valid time, the above notation can be simplified as *Key/Transaction.*

In a temporal access method it is very important to support efficiently as many types of the above mentioned queries as possible. Restricting our attention to transaction time databases, the most important and frequent temporal queries, which are addressed in the literature, are the following:

- **Key query:** Retrieve the record with $Key = K$ current at transaction time T_i (where T_i is a time instant in the time domain under consideration). For example, 'find Smith's salary as of now' ('point/point' query).
- **Range-timeslice query:** Retrieve all the records with $K_1 \leq Key \leq K_2$ that were current at transaction time T_i, for example, 'find the salary of all employees with names in the range Clark..Smith who were working at time T_i ('range/point' query).
- **Pure-timeslice (or snapshot) query:** Retrieve all the records that were current at time T_i. This is a special case of the range-timeslice operation. An example is 'find the ID's of all the employees working at time T_i' ('*/point' query).
- **Time-range query:** Retrieve all the records that were current during a time interval, for example, 'find all employee ID's working during the time interval $[T_i, T_j)$' ('*/range' query). This query generalizes the pure-timeslice query to all the time points in an interval [23].
- **General pure-key query:** Retrieve the records with $Key = K$ which were in the relation at some time during $[0, now)$. This query retrieves the whole history of the object with key K. An example is the query 'find Smith's salary history' ('point/*' query). A subclass of this type of query is the *pure-key query with a time predicate.* As an example, consider the query: 'find Smith's salary history who was employed at time T_i'.

To the authors' knowledge, none of the indexing methods which have been proposed so far, guarantees an acceptable performance for general pure-key queries (i.e. 'history of a key'). Hence, one either has to scan the whole structure sequentially, in order to detect the matching records, or to improve the structure by embedding additional structuring (e.g. pointers) and design new special algorithms. The present paper elaborates on an implementation of Overlapping B^+-trees (in the sequel: OB^+trees), designed to support transaction time in a two-dimensional key-time space. Using this structure, snapshot and range-timeslice queries can be answered optimally. However, the fundamental objective of the proposed method is to deliver exactly an efficient performance for general pure-key queries. To the authors' knowledge, the proposed structure is the first one that satisfies such queries almost optimally, i.e. linearly to the number of satisfying objects. At the same time, there is no need to maintain any additional auxiliary index. The trade-off is a small increase in time cost for version operations and storage requirements.

The remainder of the paper can be summarized as follows.The next section discusses the temporal environment and summarizes various proposals for temporal indexing. Section 3 presents briefly the original OB^+trees, the differences between the proposed method and the OB^+trees, and provides a detailed description of the new implementation. Section 4 investigates query processing in OB^+trees. Section 5 presents experimental results. Qualitative comparison with other approaches and further work are discussed in the last section.

2. Temporal access methods

2.1. Framework and assumptions

We make the following assumptions for the transaction time model. Time is assumed to be discrete and countable, though not finite, i.e. it is isomorphic to the set of integers. Hence, a value for transaction time (T_i) is a point (*instant*), lower-bounded by zero and upper-bounded by *now* (i.e. $T_i \in [0, now)$). The model implements tuple-versioning. Hence, a relation can be viewed as a set of tuples, and it is used to store the (temporal) versions of database objects. Each record has a time-invariant *key* (surrogate) and, in general, a number of time-variant attributes. For simplicity reasons, however, it is assumed that each record has exactly one time varying attribute.

A transaction time access method implicitly associates a time interval to each record. When a new record is inserted at time T_1, this time interval is set equal to $[T_1, now)$. A 'real world' deletion at time T_2 is implemented as a *logical* deletion, by changing the end_time timestamp from *now* to T_2. Updating (i.e. changing the value of the time-varying attribute of) a record at T_2 is implemented by (i) the logical deletion of the record, and (ii) the insertion of a new record (version); this new record has the same surrogate but a new attribute value. We also assume that at a specific time instant any number of modifications (insertions, deletions and updates) may take place, not just a single one.

The paper in [9] proposes a technique for batch modifications in a B^+-tree. In the present paper, we adopt the latter technique and assume that any number of update operations may take place at a specific value of transaction time. This assumption holds true especially in applications with large granularity of time (e.g. in payroll applications).

2.2. Temporal indexing and performance characteristics

In recent years a number of temporal access methods have been proposed. For a fair comparison with our transaction time access method (see Section 3), we now summarize the methods which are mostly cited in the literature and focus mainly on those which support efficient indexing on transaction time. To this end, it is noted that the fundamental objective of any temporal access method is to guarantee high performance with respect to:

- the *storage cost*, i.e. the disk storage used to physically register the data records (current and past versions), as well as the overhead for the index part of the access method,

Table 1
Parameters used for performance analysis

Parameter	Explanation
n	Total number of records in the whole structure
m	Number of records of a specific timeslice
a	Number of records satisfying the query predicate
B	Page size
S	Number of different keys ever created during the relation lifetime (in the worst case: $S = n$)

- the *update time cost*, i.e. the time needed to update the file structures when data change,
- the *query time cost*, i.e. the time needed to satisfy a complex temporal query.

The above costs are functions of the five basic parameters introduced in Table 1.

The *Monotonic B^+-tree* (MBT) [5] is built on top of modified B^+-trees and achieves a storage utilization which is close to 100%. The size of the index is very large, i.e. at the order of $O(n^2/B)$. This problem becomes more obvious when the average length of the lifespan of the indexed records becomes long. The update processing is also large, $O(n/B)$ in the worst case, whereas answering the pure-timeslice query is performed in logarithmic time, $O(\log_B n + m/B)$. By this method, range-timeslice, time-range or pure-key (general or with a time predicate) queries are inefficiently satisfied, due to the lack of clustering along both dimensions, since, the index does not organize the data by key.

The *Write Once B-tree* (WOBT) [4] has been the basis of several temporal access methods, such as the *Time Split B-tree* (TSBT) [11], the *Persistent B-tree* (PBT) [10] and the *Multi-Version B-tree* (MVBT) [1]. Basically, WOBT is a modification of the B^+-tree and performs node splits on a transaction time basis, as well as on a key value basis. The main goal of WOBT is to deliver a structure appropriate for optical disks (WORMs). However, WOBT can use either optical or magnetic disks. In the latter case, space utilization is efficient, $O(n/B)$, whereas update processing is $O(\log_B n)$. The WOBT index is well suited for pure-timeslice queries that access all the data that were valid at a given time in the past. In such a case the performance is $O(\log_B n + m/B)$.

If the WOBT is used to store a relation entirely on a WORM, it can support only a pure-key query with a time predicate: 'find Smith's salary history who was employed at time T_i. Its performance is $O(\log_B n + a)$ where, for this query, the logarithmic part corresponds to finding Smith's record at version T_i. Next, its previous a versions are accessed by using appropriate links. To answer the general pure-key query 'find Smith's salary history' requires finding whether Smith was ever an employee. The structure would need to copy 'deleted' records when a time split occurs. This implies that the WOBT state registers the most recent record for all the keys ever created. However, this increases the space consumption. If the opposite case is assumed, i.e. 'deleted' records are not copied, then all the pages including the key itself in their key space may have to be searched.

TSBT is the first method to accomplish indexing of both a (transaction) time dimension and keys under a single framework [11]. There are three kinds of splits when a data node overflows: key split, time split or key-and-time split. If most of the modifications are record insertions with a new key, then only key splits occur in the TSBT. Every timeslice query will have to visit each individual TSBT node, since all the nodes are *current*. *As of now* queries, are expected to be efficient since every node normally has many alive records. However, queries as of the remote past are estimated to be inefficient, since many of the current nodes do not contain valid records. Thus, the worst case time for a pure-timeslice query is $O(n/B)$. Accesses may also take place to pages, which are irrelevant to the query. Space overhead, update cost and pure-key queries with a time predicate have the same performance as in the case of WOBT.

For the general pure-key query, a sparse B^+-tree accompanies the TSBT, having an index entry for every different key (surrogate) ever created during the evolution of the relation. If a key has several versions, then the index record for that key points to the most recent version of the key in the TSBT structure. This index is used for the general pure-key query, because of the case that no record of that key is alive at time *now*. Thus, the query cost is $\lceil \log_B S \rceil + a$, where the first term corresponds to the cost for traversing the auxiliary B^+-tree and the second term is the number for the different versions of the key. However, the main drawback of this method is that if the user is interested in part of a key history (i.e. in a time range $[T_i, T_j)$, where $T_j < now$), then he/she is forced to access the whole key history in the structure and discard irrelevant data.

The *Snapshot Index* (SI) [21] has a space overhead $O(n/B)$ and achieves I/O cost optimality for pure-timeslice queries in $O(\log_B n + m/B)$. To answer a range-timeslice query, the whole timeslice of interest must first be computed; therefore, the answer time becomes $O(\log_B n + m/B)$. However, the basic advantage of this approach is that insertions and updates may require only a constant number of page accesses, $O(1)$, rather than the logarithmic cost required by the access methods which need also traversing the key space. The basic idea is to have only one page which accepts input data, the *acceptor page*, by using a hash index to access keys. In case that the hashing function is not a good one, then a B^+-tree can be used instead of hashing, which leads to a logarithmic worst case update performance. Similarly with TSBT, the general pure-key query is supported only by the addition of the sparse B^+-tree index. Assuming that a good hashing function is used, the pure-key query with a time predicate has a performance of $O(a)$. If a B^+-tree is used, then the cost for this type of query becomes $O(\log_B n + a)$.

R-trees [6] is the most well-known indexing structure for spatial data. When applied to temporal indexing, R-trees make use of overlapping 2-dimensional rectangles in a time-key space. The main assumption is that each object is contained in the smallest possible rectangle, called *Minimum Bounding Rectangle*. Every record is kept only once, therefore the space is linear to the number of changes, $O(n/B)$, and this is the optimal case. Since the height of the trees is $O(\log_B n)$, each record insertion requires logarithmic time. However, deletion and range timeslice search in an R-tree, imply a cost of $O(n/B)$ in the worst case. This is because the whole tree may have to be traversed, due to overlapping regions. It has been observed that records with non-uniform life-span lengths affect the update and query performance, since more than one path from the root to the leaves may have to be traversed. It is not clear how previous versions of a key can be linked together. Moreover, the general pure-key query is not addressed. Finally, another major disadvantage of R-tree based methods is that all objects should have a known in advance transaction time of deletion. In practice however this is not true, so *current* data cannot be well-accommodated in this approach. As opposed to this, the method proposed in this paper does not require such strong assumptions.

Our work concerns an efficient implementation of OB^+trees. We handle the general pure-key query almost optimally in $O(\log_B m + (T_{max} - 1) \times v) = O(vT_{max})$ time in the worst case, where T_{max} is the number of timeslices, and v denotes the maximum number of nodes involved in a split or a merge (to be explained in Section 3.2). Our design for general pure-key queries can also be incorporated in the

WOBT (for magnetic disks), the MVBT, or the TSBT structures, in order to improve their performance characteristics.

3. Overlapping B^+-trees

3.1. Description of the method

The standard B^+-tree is referred to as an 'ephemeral B^+-tree', since updates overwrite old values and only the new values are retained. In applications where old data is maintained for queries referring to the past, we might keep all the successive versions of the structure as independent B^+-trees. It is evident that each of these versions may differ from the previous or the following ones, only marginally. OB^+trees represent a transaction time indexing structure, which keeps relation timeslices indexed on transaction time. The structure of OB^+trees is produced from successive versions of B^+-trees by storing common subtrees of successive trees only once, with the maintenance of appropriate pointers in order to avoid the registration of identical (=redundant) data. Thus, none of the old values of the index are lost due to incoming updates, whereas, on the other hand, some index and data nodes in one tree can be shared by other trees. Therefore, such nodes (either internal or leaves) can be referenced by many ancestor nodes and, as a consequence, substantial storage overhead may be saved. When an update occurs, the OB^+trees structure is augmented by some information which includes:

- new leaves produced by splittings, mergings or simple updates of old leaves, and
- full paths from the root to the new leaves.

On top of the OB^+trees, we build another tree structure, to index transaction time values. We call this structure *Transaction Time Tree* (TTT).[2] Whenever a new root for transaction T_i is created we store the pair (T_i, p) in TTT, where p points to the root of the (T_i)-th structure version. We assume that this TTT structure has a reasonable size, hence it can be stored in main memory. For applications with a finer granularity of time (e.g. 10 000 structure versions) either a hashed file or a B^+-tree can be used. In the case of a B^+-tree, only its leaves have to be stored in secondary memory. Therefore, the cost for update or search of this TTT structure will not be more than a single access per root.

Note also that it is not possible to connect all the leaves of a specific ephemeral tree in a chain (as is the case in the classical B^+-tree) in order to achieve a faster answer to a range-timeslice query. The reason is that these leaves may belong to more than one instance. If this were the case, an indefinitely large number of pointers would be needed for all these lists.

Example

Fig. 1 shows how this structure was known until today [2,3,14]. More specifically, in this OB^+trees instance the index and the data node capacity equals 4. Every page contains an additional field 'reference counter' (*refcounter*), which is used to register the number of parents of a specific node. A page can be modified only if its refcounter equals 1, otherwise a copy of the page must first be allocated. The first version (Transaction time T_1) contains 21 records at the leaf level, and initially all

[2] Note that the term TTT has also been used in [17], yet with a different meaning.

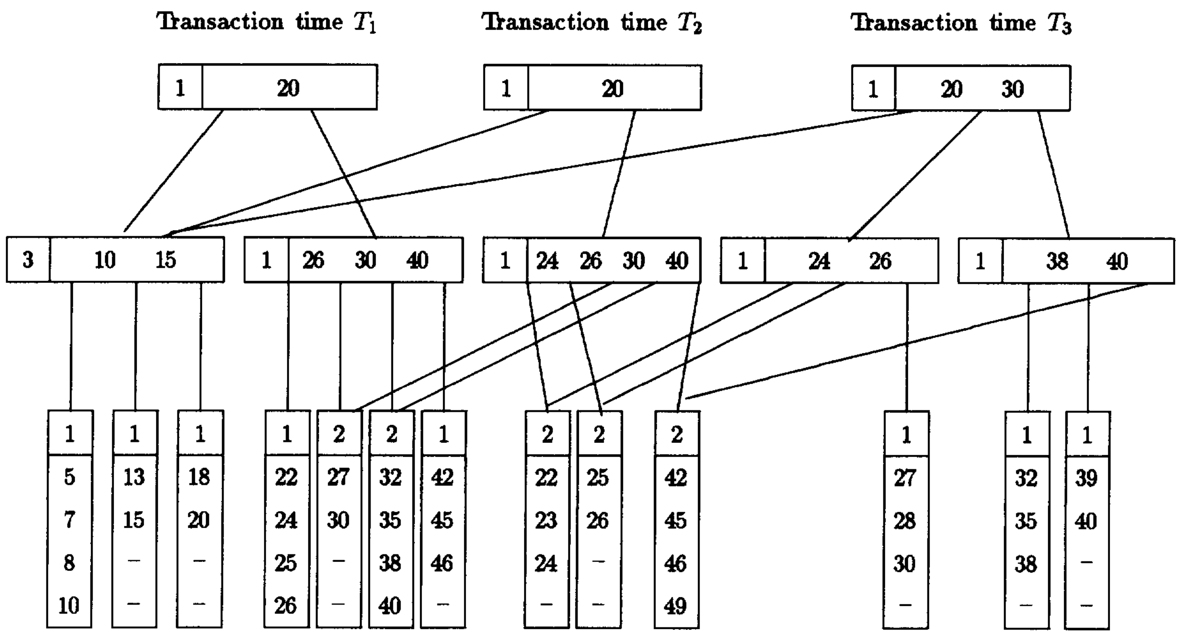


Fig. 1. The overlapping B$^+$-trees structure.

the reference counters are set equal to 1. The second version (Transaction time T_2) is produced from the first one, by the insertion of two records with keys 23 and 49. The third version (Transaction time T_3) is produced by inserting into the second version another two records, with keys 28 and 39. During the insertion the number of parents of some nodes which take active role in the procedure is increasing (refcount++). This modification causes an immediate copy of the corresponding nodes; afterwards the value of their refcounter is decreased. The reference counter values of Fig. 1 depict this final instant.

3.2. The new implementation

Here, we elaborate on OB$^+$trees aiming at providing an efficient method for searching the history of a key (i.e. support general pure-key queries) without the need to traverse more than one overlapped B$^+$-tree or to maintain some other additional separate index. In the present implementation we discard from the nodes the refcounter field of the overlapped B$^+$-trees (called *Timeslice Trees* (T-trees) in the sequel). More specifically, we replace this field by a ‘StartTime’ field that can equally well detect whether a node is being shared by other trees. However, the difference is that now we assign a value to the StartTime field during the creation of a node, hence there is no need for any further modification in the future. Moreover, we accommodate a field ‘EndTime’ to the leaves, to register the transaction time when a specific leaf becomes historical. However, the most important change is that additional pointers (e.g. see B-pointer, BC-pointer, F-pointer and FC-pointer below) are used to enhance the general pure-key query.

The internal nodes and the leaves of the T-trees have respectively the form:

```
struct {                       struct {
  int Contain;                   int Contain;
  long int Pos;                  long int Pos;
  long int StartTime;            long int StartTime;
  long int Key[MaxIn];           long int EndTime;
  long int Point[MaxIn + 1];     long int MaxKeyInNode;
} internal;                      long int Key[MaxEx];
                                 long int SecKey[MaxEx];
                                 long int B, BC,
                                   F, FC;
                               } leaf;
```

Besides the self-explanatory structures 'Key[ ]', 'Point[ ]' and the field 'Contain', the remainder fields have the following explanation:

Pos is the address of the node in the T-tree disk file.

StartTime, EndTime are, respectively, the time instants when a specific node was created, and when it became historical.

MaxKeyInNode is the value of the maximum primary key that can be stored or searched in this specific node.

SecKey is the time-variant attribute (for instance Salary) of each indexed record.

B-pointer is used during the general pure-key query in order to traverse the OB^+trees backwards. When not-null, this pointer points to a historical leaf from the previous T-tree. The node accommodating a not-null B-pointer has been created after a split, merge or update of that specific historical node.

BC-pointer is also used during the general pure-key query in order to traverse the OB^+trees backwards. The node accommodating a not-null BC-pointer is always historical. BC-pointer always refers to a historical leaf of the same T-tree. This field is involved in the merging or record redistribution procedure.

F-pointer is used during general pure-key queries in order to traverse the OB^+trees forwards. The node which accommodates a not-null F-pointer is always historical. In such a case F-pointer points to a successive T-tree leaf, which had been created from this specific historical node after a split, merge or update.

FC-pointer is also used during the general pure-key query in order to traverse the OB^+trees forwards. When this pointer is not-null, it points to a node of the same T-tree, which had been created after a split and record distribution.

It is obvious that after a batch operation with insertions, deletes and updates at a specific transaction time, we may have conceptual node splittings and mergings. In case of splittings, the F-pointer field of the historical node, points to the first of a set of new nodes which are generated after the split, whereas the B-pointer fields of the new current nodes point to the same historical data node. If we have node mergings after a batch of deletions, then the F-pointer fields of all the historical nodes involved in the merging, point to the new data node, whereas the B-pointer of the new data node, points to the first node of the historical chain. We denote by v the number of nodes involved in a conceptual splitting or merging.

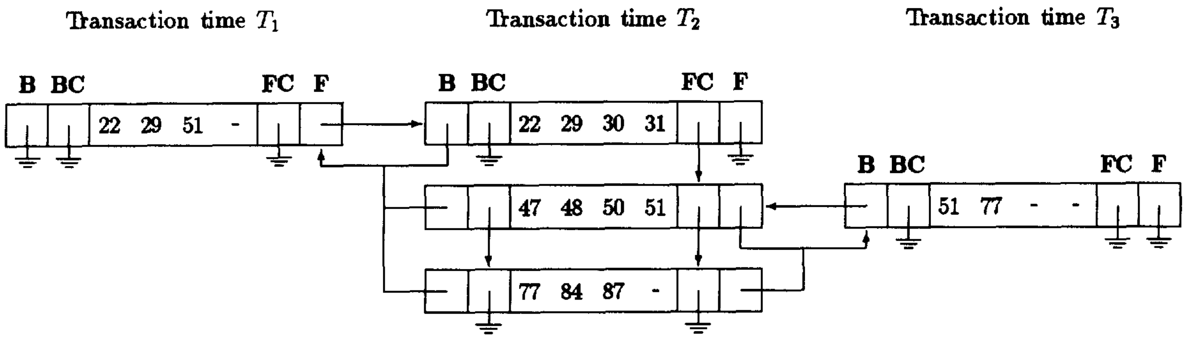


Fig. 2. Forward and backward chaining for the support of the general pure-key query.

Example

Fig. 2 shows how the leaves of three successive T-trees can be forward- and backward-chained to support the general pure-key query. A specific leaf of the T-tree is depicted on the left of the figure, corresponding to transaction time T_1. Suppose that during transaction time T_2, eight records with keys 30, 31, 47, 48, 50, 77, 84 and 87 are inserted. In such a case, we have a conceptual node split. In other words, we have to allocate three new nodes in the T-tree at transaction time T_2, in order to accommodate eleven records in total. The T-tree node of transaction time T_1 is connected to the first of these three new nodes by using the F-pointer field, whereas the FC-pointer field is used to chain all four nodes. During transaction time T_3, a set of 5 records is deleted, namely the records with key values 47, 48, 50, 84 and 87. Thus, two nodes of the T-tree of transaction time T_2 are conceptually merged to produce a new node as depicted in the figure. B-pointer and BC-pointer fields are maintained accordingly.

Some additional details should be clarified in Fig. 2. The key records are stored in the T-tree leaves, and the least key value, named 'heading key', of either a leaf or an internal node is posted to the upper level with a pointer that shows either to this specific leaf or to this internal node, respectively. During the creation of a new leaf we assign the greatest integer (in place of the theoretical ∞) to the EndTime field, in order to show when a node is still current. If during the transaction time T_i, this leaf changes from current to historical state, we set the EndTime value equal to $T_i - 1$. In the current structure therefore, it is not necessary to know beforehand the period during which a record will be active. As opposed to this, in other methods (e.g. R-tree based) which handle intervals, it is implicitly assumed that when an interval is inserted the values of both StartTime and EndTime are known in advance. Finally, T-tree index nodes are treated differently from leaves, since whenever an internal node changes to historical, no modification (=disk operation) is required.

4. Query processing in OB$^+$trees

The differences of this new OB$^+$trees version over the older one [2,3,14] are the following:

- The space increases only marginally. It is still $O(n \log_B n)$ since new pages are created for the path leading to each leaf which is updated during a new transaction time. However, there is an additional structuring via new pointer fields.

- Even though more page accesses during updates are performed, the total I/O cost per update is reduced since batch modification is applied to the structure [9]. Its performance is still $O(\log_B m)$; however, the cost per update is amortized.

These changes do have impact on the performance of the general pure-key query processing, whereas the performance of the remaining types of queries (as described in the introduction) remains the same.

OB^+trees organize data by time and key. All T-trees are maintained in a structured way, having height equal to $O(\log_B m)$. The satisfaction of a key query, a range-timeslice query, and a snapshot query is straightforward, since each timeslice has an independent root. After the appropriate root has been found, the structure is traversed by accessing pages containing only records 'alive' during the specific timeslice. In the worst case the number of transaction times is equal to the number of modification operations (i.e. insertions, deletes and updates), which is n, and the leaves of the TTT structure are stored in secondary memory. Thus, the performance cost for all these queries is $1 + \lceil \log_B m \rceil + \lceil a/B \rceil$. This way, when $n \geq m \times B$, then OB^+trees present a better performance than any other B-tree based temporal access method that has one root for several timeslices.

Until today, the OB^+trees method had to traverse all the relevant timeslices to answer a general pure-key query like 'find Smith's salary history'. Thus, the performance of such a query was not remarkable. Also, most of the other transaction time access methods do not support any type of pure-key query at all; others (such as WOBT for optical disks) support only a pure-key query with a time predicate, such as 'find Smith's salary history who was employed at time T_i', others (such as TSBT and SI) can satisfy the general pure-key query only by maintaining an auxiliary B^+-tree of height $\log_B S$ stored in secondary memory. From the above discussion it follows, that the most important advantage of the new OB^+trees implementation over other transaction time access methods is its ability to address the general pure-key query in any time range $[T_i, T_j)$ (where $0 \leq T_i$ and $T_j \leq now$), under a single framework with a very efficient performance.

4.1. Answering the general pure-key query

Our new OB^+trees version may follow any of these two methods to answer general pure-key queries:

- either start from the current T-tree (*as of now*) and use the backward chain
- or start from the first T-tree and use the forward chain.

In Appendix A, we give descriptive algorithms of both methods. In the sequel, we will give examples, to show how a general pure-key query can be satisfied.

Example

Fig. 4 depicts an example of forward and backward chaining of the leaves of three successive T-trees in an OB^+trees structure, for the support of the general pure-key query. The layout of the internal nodes and the leaves of the T-tree is illustrated in Fig. 3 (fields Point [ ] and SecKey [ ] are not shown for simplicity). Every T-tree represents an independent 'ephemeral' snapshot. The first T-tree contains 21 records at the data level. The second timeslice is the result of the batch insertion of two records, with keys 23 and 49, into the first T-tree. The third timeslice is the result of deleting from the second T-tree the record with key 30, and inserting into it a record with key 51. For simplicity, we do not depict the whole TTT structure but only the leaf level, consisting of a double linked list of three nodes.

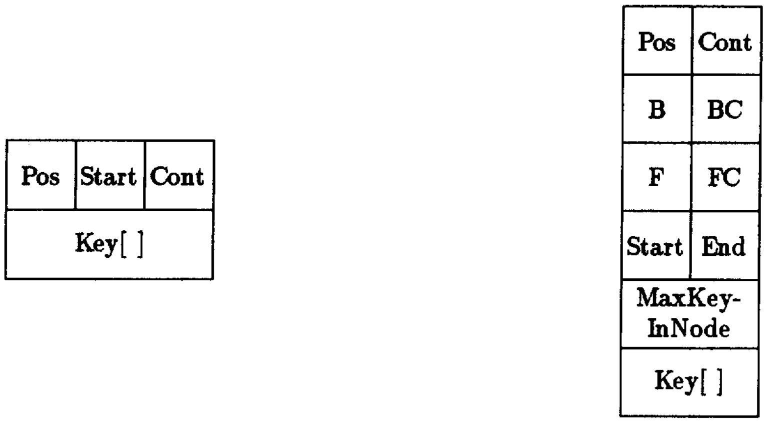


Fig. 3. Internal node and leaf node layout.

Suppose that we want to find the history of record with key 49. One possibility is to search from the most recent timeslice and move to the very first one by following the backward pointers. First, we find the root of the most recent timeslice from the last node of the TTT structure and then we traverse this timeslice until the key is found in the leaf, at Pos = 19. Then we reduce the value of the timeslice (timeslice− −) and check whether that leaf is common to the third and the second T-tree. This is achieved by comparing the StartTime field value of this node and the value of the second timeslice, that we are searching. Therefore, by using the B-pointer of that leaf, we move to the previous T-tree and find the historical leaf (Pos = 13) from which the node with Pos = 19 had been created after a

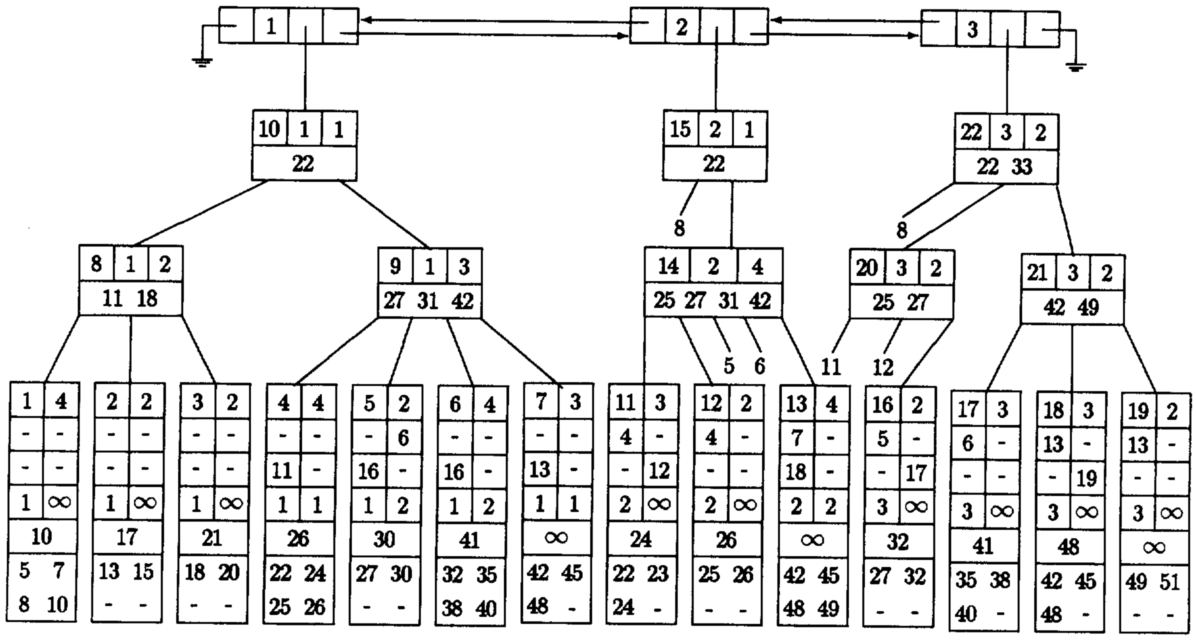


Fig. 4. An example of the new OB$^+$trees version.

split. In an analogous manner, we move to the first timeslice and access the leaf at Pos = 7, where key 49 is absent.

Another possibility is to search from the very first timeslice and move to the most recent one by following the forward pointers. First, we find the appropriate root and then we traverse this T-tree until we reach the leaf, at Pos = 7. We understand that a record with key 49 is not contained in this leaf; however, we continue by increasing the value of the timeslice (timeslice++). It is necessary to check whether that leaf is common to the first and the second T-tree, by comparing the EndTime field value of this node and the value of the second timeslice, that we are searching. We notice that the EndTime is less; by using, therefore, the F-pointer of that leaf, we move to the next T-tree (at Pos = 13) that had been created from this specific historical node after an update. The record with key 49 is contained in the leaf, at Pos = 13. By following again the F-pointer, we access the leaf at Pos = 18 of the third T-tree. At this point, we understand that the record with key 49 is not contained in this leaf. However, 49 is larger than the MaxKeyInNode value (=48) of this node. Therefore, we use the FC-pointer of this node to access the node at Pos = 19 of the same T-tree, which has a MaxKeyInNode value greater than 48. Finally, we find this record in the node at Pos = 19 of the third timeslice.

The total cost is 5 page accesses in the backward chain and 6 page accesses in the forward chain. If the B-pointer and BC-pointer (F-pointer and FC-pointer) were not available, 9 page accesses would be needed to find the history of the record with key 49, since answering the query would require traversing all the relevant T-trees.

A decision regarding the choice of the most appropriate of the above two algorithms (i.e. the algorithm with the least disk activity) can be based on the following reasoning. Suppose that T_{max} transaction times have elapsed since the creation of the initial T-tree. It is then noted that it is trivial to collect statistics about the total number of insertions and the total number of deletions which have occurred during the lifespan of the relation, e.g. from the initial T-tree until the $T_{max} + 1$ time instant (or even during certain periods of time). If the number of insertions which have taken place after the first T-tree exceeds the number of the relevant deletions, then it is obvious that the number of node splits is greater than the number of merges. In such a case, the number of nodes which are linked by FC-pointers is greater than the number of nodes which are connected via their BC-pointers. Consequently, if the number of insertions is greater than the number of deletions, then using backward pointers is more efficient than using forward pointers, and inversely.

Based on the procedures described above, we note the following. Firstly, in order to answer the general pure-key query using the backward chain (via the B-pointer and BC-pointer fields), we do not use the EndTime field of the leaves. In the hypothetical case that this field did not exist, the speed of the modification algorithm would be improved and the space overhead would be reduced. Secondly, in order to answer the general pure-key query using the forward chain (via the F-pointer and FC-pointer fields), we do not use the field StartTime of the leaves. However, we cannot remove the field StartTime from the leaves. Thirdly, we can justify the value of the field MaxKeyInNode by examining Fig. 5 more closely.

Example

When trying to find the history of the record with key 8 by using the backward chain, we first start from the third timeslice and access the leaf node with Pos = 8. By using its B-pointer, and for

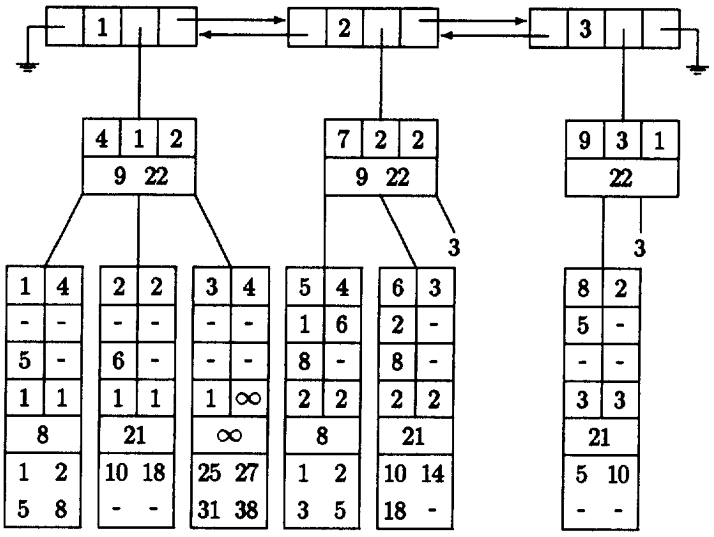


Fig. 5. Use of the field MaxKeyInNode in the new OB$^+$trees version.

timeslice 2, we access the leaf at Pos = 5, where key 8 is not present. In addition all the stored keys in this node are less than 8. If the timeslice 2 contained key 8, then this key would be registered in this node and not to its right, in leaf Pos = 6. This implies that we must stop searching at node Pos = 5, instead of accessing the node with Pos = 6 by the use of BC-pointer. The field MaxKeyInNode of the leaf with Pos = 5 helps, to this end. If it had been anticipated that the key 8 lies in the leaf with Pos = 6, then continuing the history search towards timeslice 1 by using the B-pointer of the latter node, would result in accessing the leaf with Pos = 2 rather than the correct one at Pos = 1 (where key 8 is really stored for the timeslice 1).

As mentioned earlier, the T-tree leaves are not chained. This implies that, given a general pure-key query for a range of keys, the searching procedure has to be repeated independently for each individual key. In other words, we can not just search for the first key within the range and, for every timeslice, use the BC-pointer and FC-pointer fields for the remaining keys. This is because it is not definite that these pointers will access data nodes from the timeslice currently being searched.

4.2. *Worst case performance for the general pure-key query*

Backward/forward chaining enables the OB$^+$trees to answer the general pure-key query in $O(\log_B m + a)$ time, in the best case, and in $O(\log_B m + (T_{max} - 1) \times v) = O(vT_{max})$ time in the worst case, where v ($v \ll m/B$) denotes the maximum number of nodes involved in a split or a merge. This is derived as follows.

Consider the satisfaction of a general pure-key query in the case of backward pointers (the reasoning applies similarly in the case of forward pointers). The above expression is derived by the fact that $O(\log_B m)$ page accesses are needed to traverse the current T-tree until the appropriate data node. Then we have to follow pointer B-pointer, emanating from this leaf. Assuming that the number of different versions of an existing key are a, then the best case occurs when the node which holds the key being searched, is updated only by updating a times its corresponding record (i.e. there are no node splits/merges).

The worst case may arise if the following conditions are true:

- the current leaf has been produced by merging v nodes of the previous timeslice, where these nodes are being connected via BC-pointer fields,
- the B-pointer field points to the first leaf of the specific chain of leaves mentioned before, whereas
- the key we are seeking resides in the last leaf within the same set of leaves.

If all these conditions are satisfied, then we perform v accesses to find the previous version of the object in question. By generalizing for all timeslices the term $(T_{max} - 1) \times v$ is derived. It is reasonable to accept that the first term (i.e. $\log_B m$) does not contribute to the total cost as the second term does. Finally, therefore, the cost for answering a general pure-key query is $O(vT_{max})$, in the worst case. At this point, note that the relation $vT_{max} \geqslant a$ is satisfied.

Example

Suppose that we search for the history of the record with key 44, in the OB$^+$tree of Fig. 6. Assume also that we start from the last timeslice and use the backward method. In the worst case, the B-pointer of the last leaf points to the first leaf of the previous T-tree, whereas the key that we search for, is in the last leaf of the latter timeslice. Alternatively, as can be understood by a simple inspection of Fig. 6, if we had tried to find the history of key 44 by traversing independently the three T-trees, then the total number of page accesses would be 6, i.e. less than the cost of using the backward method (= 8 page accesses). Analogous cases may arise when the forward method is applied.

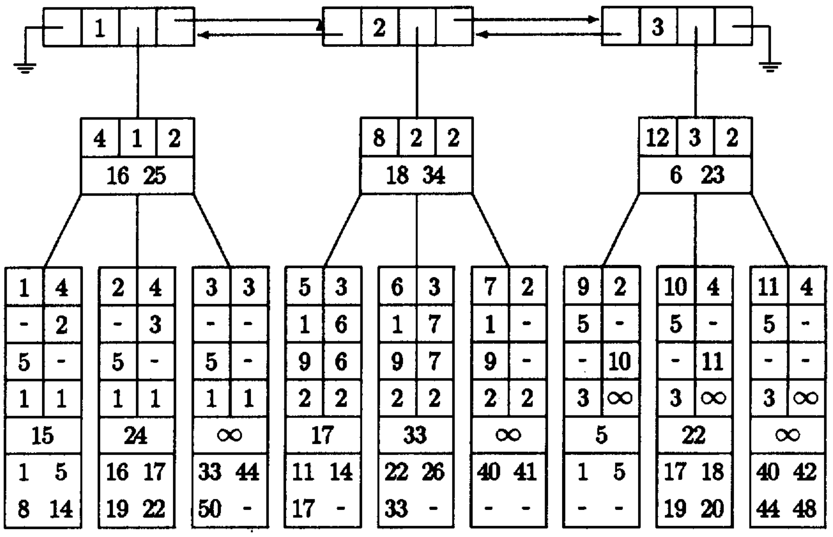


Fig. 6. Worst case performance for the general pure-key query using the backward (forward) chaining.

5. Experiments

We implemented the OB$^+$trees structure in C, and experimented with it by using parameter values as follows. Assuming that the page size is 2048 bytes, the pointer size, as well as the size of the StartTime and the EndTime fields are 4 bytes each, we conclude that the internal nodes of the T-tree can accommodate 250 (*key,pointer*) pairs. We also assume that the transaction time values range from 0 to $T_{max} = 10$. Therefore, the TTT structure has a small size and can be stored in main memory. It is obvious that if the time domain is very large, then we can use a disk-resident *Transaction Time B^+-tree*. In all the experiments, we considered a blocking factor MaxEx of 20, 10 and 4 records. Also, we assumed that the input data are sorted in increasing order, so as to make use of a combined algorithm which performs batch insertions, deletions and updates, along the lines of [9]. This batching technique can be implemented by buffering a certain number of insertions, deletions and updates occurring during a transaction time, and sorting them in main memory with a negligible cost. Table 2 contains a list of the parameters used and their values, as well. A common conclusion in all the experiments is that the OB$^+$trees with the least MaxEx value has the greatest I/O cost and storage requirements.

5.1. Storage requirements

5.1.1. First experiment

In total, we inserted $n = 10\,000$ keys in an empty structure. More specifically, during each transaction time we inserted 1000 records with keys in increasing order (batch insertions). The main goals were to count the total number of page accesses and measure the increase of disk space, as a function of transaction time. Fig. 7 demonstrates the behavior of these two performance Metrics, for various MaxEx values.

The abrupt increase of the number of I/O's in the left part of Fig. 7, after the first transaction time is explained by the fact that at this point in time there was not any initial T-tree. Thus, there was no need to read historical nodes for time splits, or any need to assign the F- or the BC-pointer field of any historical leaf node.

Table 2
Parameter values used for experimentation

Parameter	Value(s)	Comments
B	2048	Equal to T-tree node size
T_{max}	10	Maximum transaction time instant
n	10 000	Total number of records in the whole structure
MaxIn	250	Maximum capacity in a T-tree internal node
MinIn	MaxIn/2	Minimum capacity in a T-tree internal node
MaxEx	4, 10, 20	Maximum capacity in a T-tree data node
MinEx	MaxEx/2	Minimum capacity in a T-tree data node
m	0 ... 10 000	Number of records in a specific T-tree

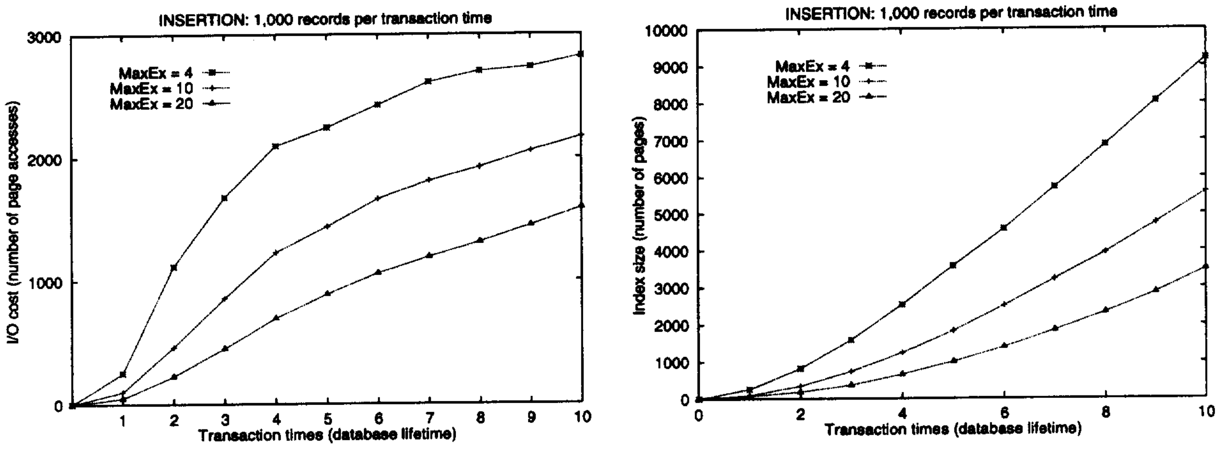


Fig. 7. Batch insertions: I/O cost and disk space used as a function of transaction time.

5.1.2. Second experiment

At the beginning the current T-tree contained $m = n = 10\,000$ 'alive' records. During 10 transaction times we deleted all these keys, 1000 records per transaction time, by assuming that deletions were in increasing key-value order (batch deletions). The main goals were to count the total number of disk operations and to measure the increase of disk space, as a function of transaction time. Fig. 8 demonstrates the behavior of these two performance metrics for various MaxEx values.

5.1.3. Third experiment

At the beginning the current T-tree contained $m = n = 10\,000$ 'alive' records. During 10 transaction times we inserted, updated and deleted a total of 3000 records. More specifically, at each transaction time we inserted, updated and deleted 100 records respectively, with keys in increasing order (batch modifications). Evidently, the structure finally contained $m = 10\,000$ records of 'alive' keys in the current (as of *now*) T-tree. We note that only 30% of the records accommodated in the most current

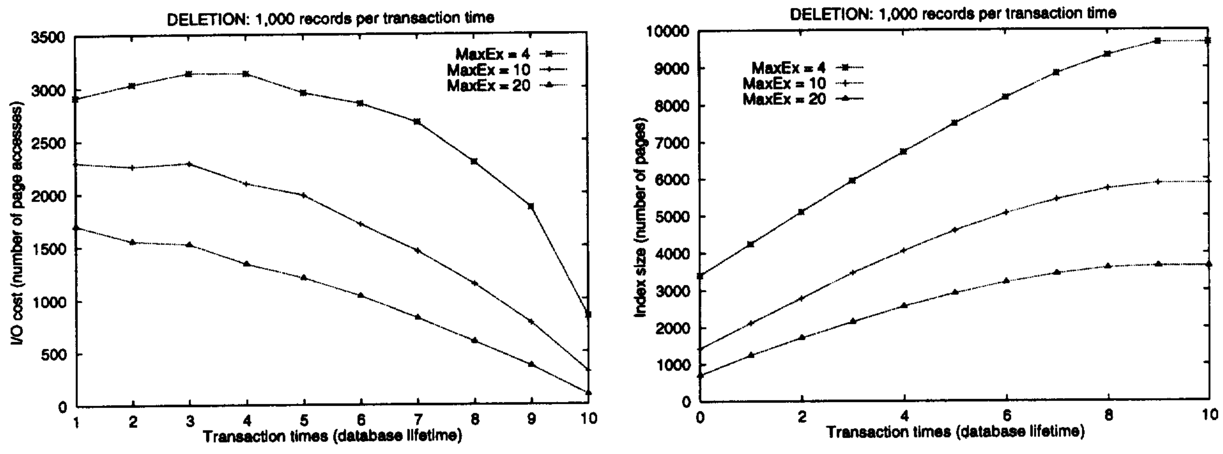
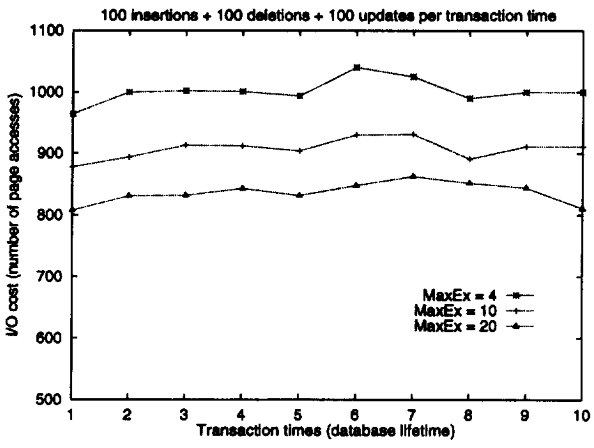


Fig. 8. Batch deletions: I/O cost and disk space used as a function of transaction time.



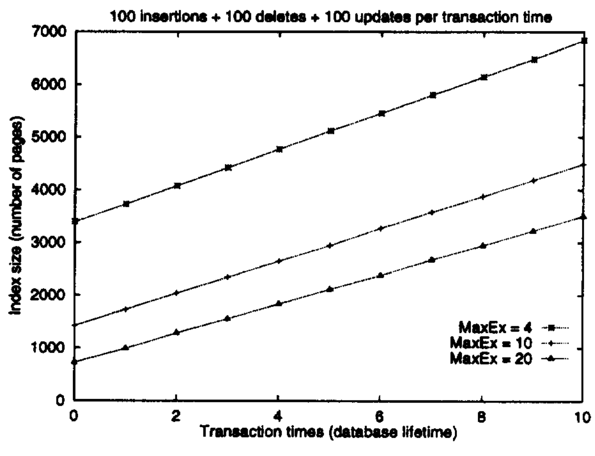


Fig. 9. Batch insertion, deletion and update of 3000 records: I/O cost and disk space used as a function of transaction time.

T-tree is different than the records in the initial T-tree. This means that the lifespan of the remaining 70% of the initial records equals [0, 10). Fig. 9 demonstrates the behavior of the total I/O activity and the disk space increase as a function of transaction time, for various MaxEx values. From the latter figure it appears that the I/O cost per transaction time is almost stable, whereas the final index size is about 4, 3 and 2 times larger than the initial size, for MaxEx values equal to 20, 10 and 4, respectively. This fact is explained by taking in consideration the percentage of identical information that is being copied from a specific timeslice to the next one.

5.2. Query processing time

We did not investigate the I/O performance to process the pure-timeslice or the range-timeslice queries, because they are satisfied by traversing a single T-tree. Since T-trees are almost typical B^+-trees, the pure-timeslice and range-timeslice queries have the well-known logarithmic I/O performance, $O(\log_B m + m/B)$ and $O(\log_B m + a/B)$, respectively. Hence, we did not have to experiment with these queries.

Next, we investigated the I/O performance of the general pure-key query. For this experiment, we used the structure produced at the end of the third experiment of the previous subsection. Recall that this structure has 11 timeslices ($T_i = 0$ to 10) with $m = 10\,000$ 'alive' records per any T-tree. We searched for the history (general pure-key query) of a total number of 100 keys. The keys were chosen in a random way, i.e. it was not known beforehand whether a key really exists or not. In order to study the performance of the general pure-key query as a function of the range of the time predicate, two experiments were carried out, described in the next two paragraphs.

5.2.1. First experiment

In the first experiment we made use of the backward pointers. More specifically, we started from the current timeslice (i.e. $T_{max} = 10$) and tried to answer the general pure-key query for 100 random keys, by searching in one timeslice, i.e. the last one. Next, we started again from the current timeslice and tried to answer the general pure-key query for the same 100 keys, going backwards to the previous timeslice, i.e. by searching in two timeslices. We continued this way until we searched the history of the same 100 keys within the whole lifespan of the relation. Each time, we counted the total number of page accesses per key. For instance, searching in 11 timeslices, we counted per key 6.88 accesses for MaxEx = 20, 5.37 accesses for MaxEx = 10, and 4.02 accesses for MaxEx = 4.

5.2.2. Second experiment

In the second experiment we followed the reverse way of searching, i.e. we tried to answer the general pure-key query for the same 100 keys as in the previous experiment, but we moved forward, starting from the first timeslice. Again, we increased successively the number of timeslices involved, starting from one and ending up with 11 timeslices. Each time we counted the total number of page accesses per key. This way, in 11 timeslices we counted per key 6.87 page accesses for MaxEx = 20, 5.36 accesses for MaxEx = 10, and 4.07 accesses for MaxEx = 4.

The plots in Fig. 10 illustrate the results of the above two experiments. These plots show how the I/O activity increases as a function of the number of the timeslices being searched. The x axis depicts the total number of searched timeslices, whereas the y axis gives the total number of I/O's per key. The three curves in each plot correspond to the three MaxEx values mentioned above. It should be noted that in the case of a general pure-key query, the I/O operations, to search a key in only one timeslice, correspond to the accessed nodes during the traversal of a specific T-tree. Thus, in Fig. 10 we observe that all the curves start from point $(x, y) = (1, 3)$, which is the height of a single T-tree.

As expected, the curves for the same MaxEx in the two figures are quite similar, since the number of insertions and deletions carried out during the third experiment was the same (i.e. 1000). Hence, we cannot take advantage of the technique described in Section 4, concerning the choice of backward or forward chaining in order to achieve the best performance. Thus, both algorithms behave equivalently.

For the case of the general pure-key query, we also notice in the two plots the smooth increase of the total number of I/O's per key with respect to the increase of the number of timeslices involved in the query. For example, starting from the last timeslice and going backwards until the first T-tree, with respect to the MaxEx values, we counted 6.46, 5.09 and 3.93 page accesses per key for searching in

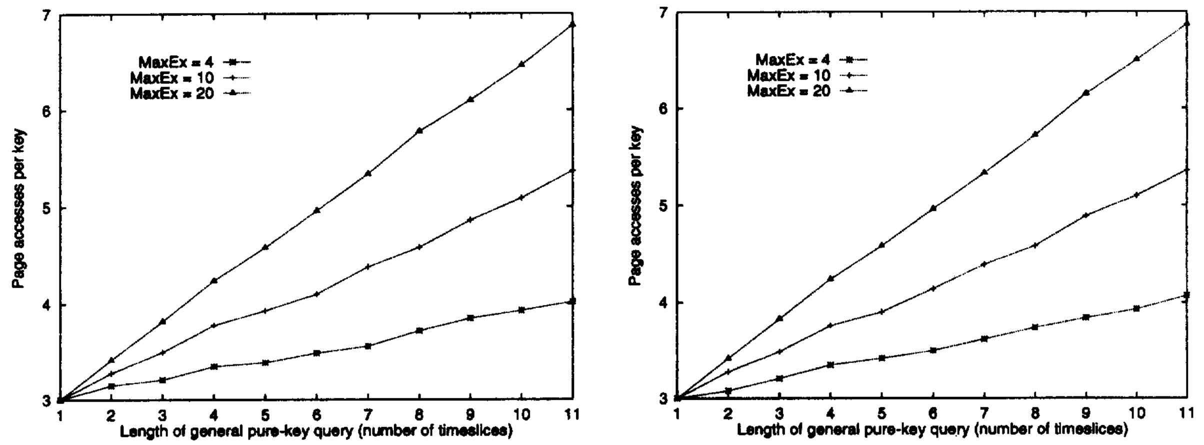


Fig. 10. Search for the history of 100 keys: Backward and forward.

10 timeslices, whereas we counted 6.88, 5.37 and 4.02 accesses per key when searching in 11 timeslices. In the special case of MaxEx = 4, we note that, interestingly, the two lower curves in Fig. 10 are almost constant functions, since for backward searching we counted 3.72 I/O's for 8 timeslices, 3.85 I/O's for 9 timeslices, 3.93 I/O's for 10 timeslices, 4.02 I/O's for 11 timeslices, and so on. Thus, the increase was very small each time, close to 0.1.

With respect to the relationship between the performance of general pure-key queries and the value of the blocking factor MaxEx, we notice the following. Evidently, as the number of stored records per leaf decreases (relevant to the decrease of MaxEx values), the number of leaves in each T-tree increases. The greater the number of the T-tree data nodes, the more rare it is for a specific leaf to be updated. This means that the greater the number of T-tree data nodes is, the greater the number of leaves with records having 'long lifespans'. During the third experiment of the previous subsection, we noticed that every node has been updated in the worst case only twice. This is the reason why the query cost of OB^+trees is lower for lower MaxEx values.

It is also noticed that the lower the MaxEx value is, the greater is the total number of OB^+trees data nodes. This is explained by the fact that the lower is the maximum number of stored keys per node, the greater is the number of node updates produced at each transaction time. Thus, the total number of data nodes in the structure increases faster. This way, when we search for a key or the history of a key, the time cost to locate a specific page increases. In order, therefore, to achieve a general improvement of the time cost (i.e. both maintenance and query cost), the appropriate MaxEx value must be carefully chosen.

6. Summary and future work

We presented a new efficient version of the OB^+trees indexing technique. OB^+trees organize data both on a transaction time basis and on the key space basis, using a two level storage structure. The new approach includes the storage requirements, the update characteristics and the query performance against three basic temporal queries: the general pure-key, the pure-timeslice and the range-timeslice queries.

Table 3 summarizes the performance characteristics of this OB^+trees version, whereas Table 4 gives a qualitative performance comparison between OB^+trees and other relevant transaction time indexing methods. 'N/A' denotes that this query is *Not Applicable* in the structure, unless the latter is exhaustively searched to satisfy the query. For the case of a pure-key query with a time predicate case satisfied by the Snapshot Index, we assume that a good hashing function is being used. If a B^+-tree is used, then its performance becomes worse i.e. $O(\log_B n + a)$.

Based on these results, we conclude that this new OB^+trees version is superior than most other transaction time access methods, yet at the cost of additional storage requirements. Using efficiently

Table 3
Performance characteristics of the new OB^+trees version

Total space overhead	Update per change	Pure-timeslice query	Range-timeslice query	General pure-key query	Additional auxiliary index
$O(n \log_B n)$	$O(\log_B m)$	$O(\log_B m + m/B)$	$O(\log_B m + a/B)$	$O(vT_{max})$	No

Table 4
Performance characteristics of other transaction time access methods

Access method	Space overhead	Update per change	Pure-timeslice query	Range-timeslice query	Pure-key query with a time pred.	General pure-key query
MBT [5]	$O(n^2/B)$	$O(n/B)$	$O(\log_B n + m/B)$	$O(\log_B n + m/B)$	N/A	N/A
WOBT [4]	$O(n/B)$	$O(\log_B n)$	$O(\log_B n + m/B)$	$O(\log_B n + a/B)$	$O(\log_B n + a)$	N/A
TSBT [11]	$O(n/B)$	$O(\log_B n)$	$O(n/B)$	$O(n/B)$	$O(\log_B n + a)$	$O(\log_B S + a)$
SI [21]	$O(n/B)$	$O(1)$	$O(\log_B n + m/B)$	$O(\log_B n + m/B)$	$O(a)$	$O(\log_B S + a)$
R-trees [6]	$O(n/B)$	$O(\log_B n)$	$O(n/B)$	$O(n/B)$	N/A	N/A
MVBT [1]	$O(n/B)$	$O(\log_B m)$	$O(\log_B n + m/B)$	$O(\log_B n + a/B)$	$O(\log_B n + a)$	$O(\log_B S + a)$

both forward and backward chaining of historical data nodes (and *not* of different versions of keys), the new structure achieves a very efficient performance in the case of the general pure-key query without any time overhead such as query modification (for instance, pure-key query with a time predicate) and without the use of secondary indexes. Most of the other transaction time access methods do not support any type of pure-key query at all; some others (such as WOBT for optical disks) support only the pure-key query: 'find Smith's salary history who was employed at time T_i', that is a time predicate has to be provided in the query; other access methods (such as TSBT and SI) can support the general pure-key query only by maintaining an additional auxiliary tree index stored in secondary memory, as depicted in Table 5. Our design for general pure-key queries can also be incorporated in other designs such as WOBT for magnetic disks, MVBT or TSBT structures to support efficiently this temporal query.

Further work includes experimental comparison of the performance of the OB^+trees with that of other transaction time access methods, especially those that are modifications of the B-tree family (e.g. WOBT, TSBT, SI, MVBT, etc). Such a piece of work would aim at establishing a reliable benchmark for TDBs. Another piece of work is a bi-temporal extension to the proposed OB^+trees, to support both valid and transaction time for temporal databases. This can be achieved by a careful replacement of the T-trees by MAP21 trees [15]. In such a case we will not index keys in the OB^+trees data nodes, but points of valid time ranges which overlap between successive transaction times. OB^+trees have been used to store spatio-temporal data (i.e. raster images) in [22] and further research is being performed toward efficient spatio-temporal window query processing.

Table 5
Alternative structures with/without additional auxiliary indexes to support the general pure-key query

Access method	Additional auxiliary index	Additional space overhead	Additional access per change
MBT [5]	No	–	–
WOBT [4]	No	–	–
TSBT [11]	Yes	$O(S/B)$	$O(\log_B S)$
SI [21]	Yes	$O(S/B)$	$O(\log_B S)$
R-trees [6]	No	–	–
MVBT [1]	Yes	$O(S/B)$	$O(\log_B S)$

Acknowledgments

The authors would like to thank Mr. Apostolos Papadopoulos for his suggestions in the course of experimentation. Research performed under the European Union's TMR Chorochronos project, Contract No. ERBFMRX-CT96-0056 (DG12-BDCN).

Appendix A

Here we provide the algorithms for answering a general pure-key query by using the backward or the forward technique.

```
Procedure GeneralPureKeyQueryBackwardChain(key k)
Begin
timeslice = lastInDataBase;   /* lastInDataBase = $T_{max}$ */
find the root of the timeslice using the TTT structure;
traverse the T-tree(timeslice) downto the appropriate leaf q;
printInfoAboutTheKey(timeslice,q,k);   /* either the key k exists or not in this timeslice */
while(q.StartTime $\leqslant$ --timeslice)
  printInfoAboutTheKey(timeslice,q,k);   /* q is always the same—no extra I/O */
while(timeslice $\geqslant$ firstInDataBase)   /* firstInDataBase = 0 */
  Begin
    q = q.B;
    while(k < MaxKeyInNode(q))
      q = q.BC;
      printInfoAboutTheKey(timeslice,q,k);   /* either k exists or not in this timeslice */
      while(q.StartTime $\leqslant$ --timeslice)
      printInfoAboutTheKey(timeslice,q,k);   /* q is always the same—no extra I/O */
  End;  /* while */
End;  /* void */
```

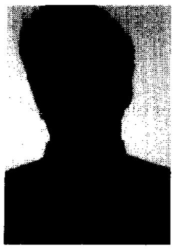
```
Procedure GeneralPureKeyQueryForwardChain(key k)
Begin
timeslice = firstInDataBase;   /* firstInDataBase = 0 */
find the root of the timeslice using the TTT structure;
traverse the T-tree(timeslice) till the appropriate leaf q;
printInfoAboutTheKey(timeslice,q,k);   /* either the key k exists or not in this timeslice */
while(q.EndTime $\geqslant$ ++timeslice)
  printInfoAboutTheKey(timeslice,q,k);   /* q is always the same—no extra I/O */
while(timeslice $\leqslant$ lastInDataBase)   /* lastInDataBase = $T_{max}$ */
  Begin
    q = q.F;
    while(k < MaxKeyInNode(q))
      q = q.FC;
    printInfoAboutTheKey(timeslice,q,k);   /* either k exists or not in this timeslice */
    while(q.EndTime ≥ ++timeslice)
      printInfoAboutTheKey(timeslice,q,k);   /* q is always the same—no extra I/O */
  End;  /* while */
End;  /* void */
```

References

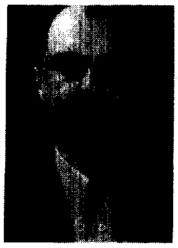
[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, On optimal multi-version access structures, *The VLDB Journal* 5(4) (1996) 264–275.
[2] F.W. Burton, M.W. Huntbach, J. Kollias, Multiple generation text files using overlapping tree structures, *The Computer Journal* 28(4) (1985) 414–416.
[3] F.W. Burton, J.G. Kollias, V.G. Kollias, D.G. Matsakis, Implementation of overlapping B-trees for time and space efficient representation of collection of similar files, *The Computer Journal* 33(3) (1990) 279–280.
[4] M.C. Easton, Key-sequence data sets on indelible storage, *IBM Journal on Development* 30(3) (1986) 230–241.
[5] R. Elmasri, M. Jaseemuddin, V. Kouramajian, Partition of time index for optical disks, *Proc. 8th IEEE International Conference on Data Engineering*, Phoenix, AZ (1992) pp. 574–583.
[6] A. Guttman, R-trees—A dynamic index structure for spatial searching, *Proc. ACM SIGMOD Conf.*, Boston, MA (1984) pp. 47–57.
[7] C.S. Jensen, J. Clifford, R. Elmasri, S. Gadia, P. Hayes, S. Jajodia, C. Dyreson, F. Grandi, W. Kafer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. Roddick, N. Sarda, M. Scalas, A. Segev, R. Snodgrass, M. Soo, A. Tansel, P. Tiberio, G. Wiederhold (Eds.), A consensus glossary of temporal database concepts, *ACM SIGMOD Record* 23(1) (1994) 52–64.
[8] A. Kumar, V.J. Tsotras, C. Faloutsos, Designing access methods for bi-temporal databases, *IEEE Trans. on Knowledge and Data Engineering* 10(1) (1998) 1–20.
[9] SD. Lang, J.R. Driscoll, Improving the differential file technique via batch operations for tree structured file organizations, Proc. IEEE Int. Conf. on Data Engineering, Los Angeles, CA (1986).
[10] S. Lanka and E. Mays, Fully persistent B^+-trees, *Proc. ACM SIGMOD Conf.*, Denver, CO (1991) pp. 426–435.
[11] D. Lomet, B. Salzberg, Transaction time databases. In: A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass (Eds.), *Temporal Databases: Theory, Design and Implementation*, Benjamin/Cummings (1993) pp. 388–417.
[12] N.A. Lorentzos, Y. Manolopoulos, Functional requirements for historical and interval extensions to the relational model, *Data and Knowledge Engineering*, 17 (1995) 59–86.
[13] N.A. Lorentzos, Y.G. Mitsopoulos, SQL extension for interval data, *IEEE Trans. on Knowledge and Data Engineering* 9(3) (1997) 480–499.
[14] Y. Manolopoulos, G. Kapetanakis, Overlapping B^+-trees for temporal data, *Proc. 5th JCIT Conf.*, Jerusalem, Israel (October 1990) pp. 491–498.
[15] M. Nascimento, Efficient indexing of temporal databases via B^+-trees, Ph.D. Dissertation, Technical Report, Southern Methodist University, 1995; http://www.dcc.unicamp.br/~mario/papers.html/dissertation.ps.gz
[16] M. Nascimento, M. Eich, An introductory survey to indexing techniques for temporal databases, Technical Report, Southern Methodist University, 1995; http://www.dcc.unicamp.br/~mario/papers.html/tr-95-cse-01.ps.gz
[17] M. Nascimento, M. Eich, R. Elmasri, M-IVTT—An Index for Bi-temporal databases, *Proc. 7th DEXA Conf., Lecture Notes in Computer Science*, Vol. 1134, Springer Verlag, Zurich, Switzerland (1996) pp. 779–790.
[18] B. Saltzberg, V. Tsotras, A comparison of access methods for time evolving data, *ACM Computing Surveys*, in press; ftp://ftp.ccs.neu.edu/pub/people/salzberg/tempsurvey.ps.gz
[19] R.T. Snodgrass, I. Ahn, Temporal Databases, *IEEE Computer* 19(9) (1986) 35–42.
[20] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, R. Snodgrass (Eds.), *Temporal Databases: Theory, Design and Implementation*, Benjamin/Cummings (1993).
[21] V. Tsotras, N. Kangelaris, The Snapshot Index—An I/O optimal access method for timeslice queries, *Information Systems* 20(3) (1995) 237–260.

[22] T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos, Overlapping linear quadtrees—A spatiotemporal access method, *Proc. 6th ACM Workshop on Advances in Geographical Information Systems (ACM-GIS 98)*, pp. 1–7, Bethesda, MD (1998).

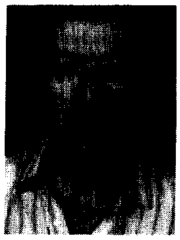
[23] P.J. Varman, R.M. Verma, An efficient multi-version access structure, *IEEE Trans. on Knowledge and Data Engineering* 9(3) (1997) 391–409.



Theodoros Tzouramanis received a 5-year B.Eng. (1996) in Electrical and Computer Engineering from the Aristotle University of Thessaloniki. He is currently a doctoral student in the Department of Informatics of the same university. His main research interests are access methods and query processing for spatio-temporal databases.



Yannis Manolopoulos received a 5-year B.Eng. (1981) in Electrical Engineering and a Ph.D. (1986) in Computer Engineering, both from the Aristotle University of Thessaloniki. He has been with the Department of Computer Science of the university of Toronto, the Department of Computer Science of the University of Maryland at College Park and the Department of Electrical and Computer Engineering of the Aristotle University of Thessaloniki. Currently, he is Associate Professor at the Department of Informatics of the latter university. He has published over 80 papers in refereed scientific journals and conference proceedings. He is the author of two textbooks on data/file structures, which are being taught in most of the computer science/engineering departments in Greece. He is a member of the Greek Computer Society, Technical Chamber of Greece, ACM and IEEE Computer Society. His research interests include spatial and temporal databases, geographical information systems, www-based information systems, data mining, text databases, and performance evaluation of storage subsystems.



Nikos A. Lorentzos received a B.Sc. in Mathematics from University of Athens, an M.Sc. in Computer Science from Queens College, CUNY and a Ph.D. in Computer Science from Birkbeck College, London University. He is an Assistant Professor at the Agricultural University of Athens. He was the technical manager of the ORES project (ESPRIT III), which was concerned with, among other things, the design and implementation of VT-SQL, a valid time extension to SQL. He is or has been involved successfully in projects funded by either the European Union or the Greek public sector. Citations to his research work exceed 100, some of them with positive criticisms. He acts as a reviewer for major research journals and international conferences and as an evaluator of research projects for the European Union. He is a member of the Greek Mathematics Society, Greek Computer Society, BCS and ACM. His research interests include, among others, data modeling, databases, expert systems and geographical information systems.